第５章　MPI_c ライブラリーの仕様

5.1 はじめに

本ライブラリーMPI_c (message-passing-interface_compact)は並列処理用通信ライブラリーであり、主要なメッセージパッシングライブラリーの規格である MPI の一部の機能を実装したものである。MPI_c は、並列処理に必要な初期化･終了、送受信、状態取得など基本的な機能を有する。以後このライブラリーを MPI_c と呼ぶ。

MPI_c は Windows 環境でのみで動作し、呼び出し側の対象プログラム言語は C++と Fortran である。Fortran 版の関数は Compaq 社の Visual Fortran で Fortran90 を用いた場合のみ動作する。また、並列システムを構成するコンピュータは､TCP/IP の使用可能なネットワーク環境で動作していることが必須である。現在のところ、MPI_c はこれらの条件を満たす環境で動作確認を行っている。

5.2 MPI_c 開発環境

並列処理用通信ライブラリーの開発環境として、MPI_c ライブラリーは mpi_c project フォルダーに収められている。そのフォルダー内における構成が図 5-1 に示される。

図 5-1 MPI_c の構成

mpi_c project フォルダー内の各フォルダーの説明を以下に記す。

mpi_c フォルダー：

MPI_c に関するソースプログラムの全てが収められている。従って、MPI_c の機能を拡張したい場合や変更を加えたい場合、あるいは実装の詳細を知りたいときはこのフォルダー内のソースコードを参照することになる。

mpi_c_deamon フォルダー：

このフォルダー内には、常駐プログラム mpi_c_deamon のソースコードが収められている。mpi_c デーモンは、マスター側からの要求を受けて、スレーブ用の動的解析プログラムを動作させる。

mpi_c_dll フォルダー：

MPI_c を利用したアプリケーションで必要となる DLL ファイルが収められている。

sample フォルダー：

MPI_c ライブラリーを用いた簡単なサンプルアプリケーションが収められている。サンプルは C++用と Fortran 用のサンプルがあり、マスター・スレーブモデルで構築されている。

バージョンの概要.txt：

このファイルは、MPI_c ライブラリーに関する現在のバージョンの概要と過去のバージョンの変更履歴が書かれたテキストファイルである。

5.3 MPI_c の使用方法

5.3.1 C++で使用する場合

通信ライブラリーMPI_c を並列アプリケーション内で使用したい場合は以下の手順に従う必要がある。ただし、手順は、このライブラリーをコールする言語が C++の場合と Fortran の場合とでは多少異なるので、ここでは各々の場合に分けて説明する。最初に、C++で使用する場合について行う。なお、開発環境を Microsoft 社の Visual Studio とする。

図 5-2 mpi_c_dll フォルダーの内容

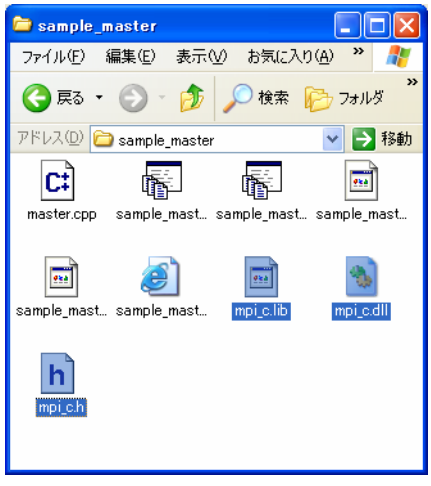

図 5-3　必要ファイルのコピー

1）まず、MPI_c を使用するアプリケーションのフォルダー内に、MPI_c project 内の「mpi_c_dll」(図 5-2)フォルダーの mpi_c.dll、

mpi_c.lib、mpi_c.h ファイルをコピーする(図 5-3 の選択状態のファイル)。

２) 次に Visual Studio 内の「プロジェクト」メニューの「設定」を選択すると図 5-4 に示すダイアログが現れる。そのダイアログ内で「リンク」タブを選択し、「オブジェクト／ライブラリ／モジュール」項目に「mpi_c.lib」を追加する(図 5-4 の選択状態部分)。

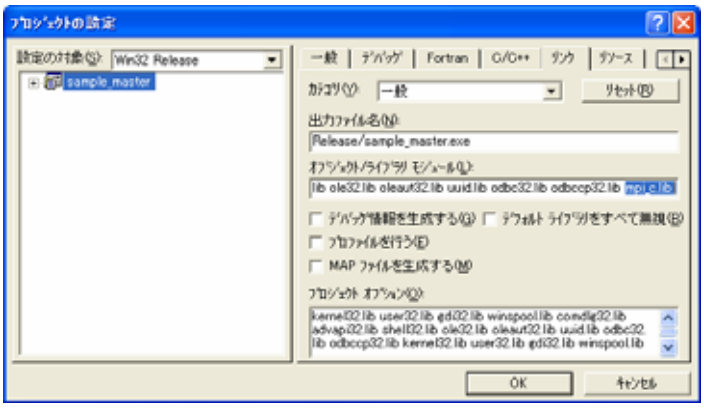

図 5-4　mpi_c.lib の追加

３)最後に、実際のプログラム内で、mpi_c 関数を使用したい箇所より手前の部分で「mpi_c.h」をインクルードすれば C++内で MPI_c を使用することができる(図 5-5)。

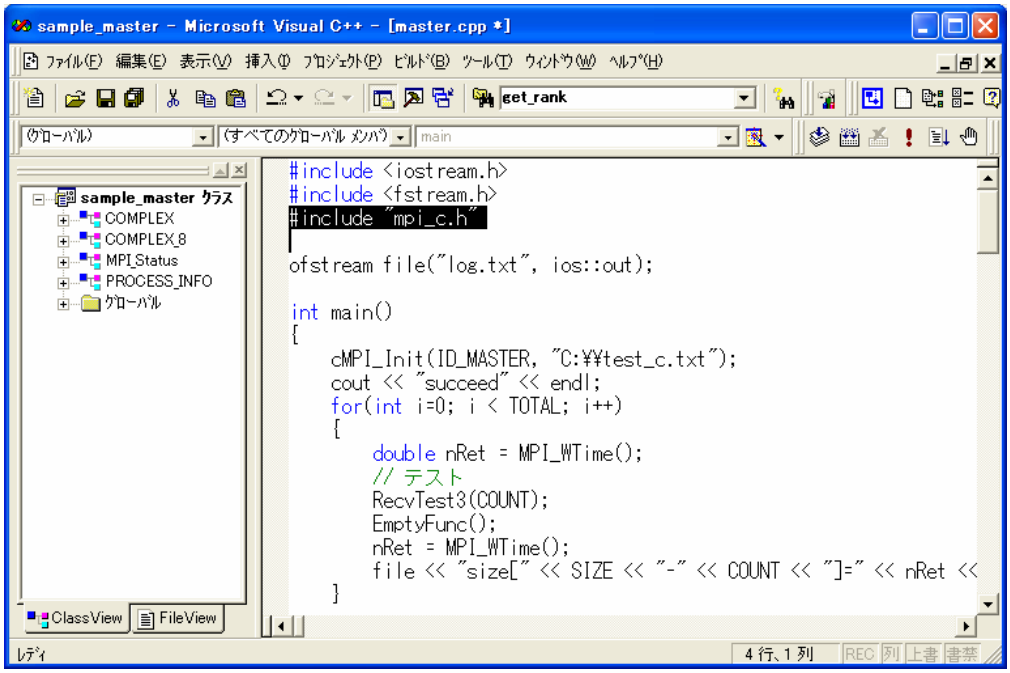

図 5-5　mpi_c.h のインクルード

5.3.2 Fortran で使用する場合

本節では、Fortran を用いて、ライブラリーMPI_c を使用する場合の手順について解説する。なお、開発環境は Conpaq 社の Visual Fortran を用いるものとする。

１）最初に、使用したいアプリケーションのフォルダー内に、mpi_c project 内の「mpi_c_dll」(図 5-2)フォルダーの mpi_c.dll、mpi_c.lib、mpi_c.h ファイルと **mpi_c_fortran** フォルダーをコピーする(図 5-6 の選択状態のファイル)。

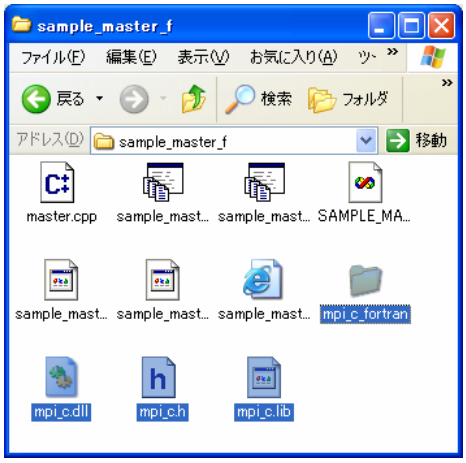

図 5-6　必要ファイル(フォルダ)のコピー

２）C++での使用手順の 2)と同様であるが、「プロジェクト」メニューの「設定」を選択すると図 5-4 に示すダイアログが現れる。そのダイアログ内で「リンク」タブを選択し、「オブジェクト／ライブラリ／モジュール」項目に「mpi_c.lib」を追加する(図 5-4 の選択状態部分)。

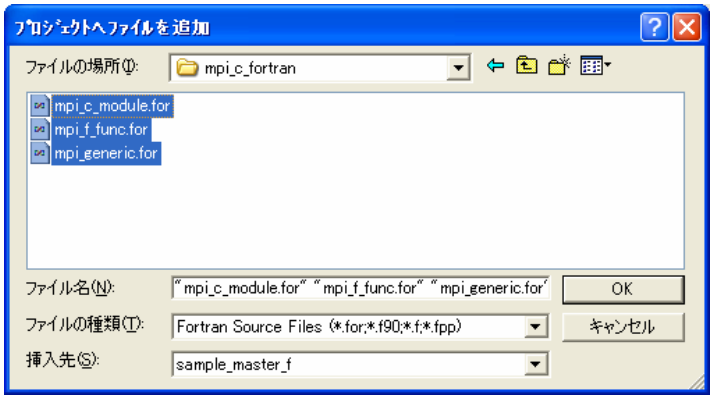

図 5-7　MPI_c 用 fortran ファイルのプロジェクトへの追加

３）次に、「プロジェクト」メニューで「プロジェクトに追加」→「ファイル」を選択すると、図 5-7 に示すダイアログが現れる。ここで、「mpi_c_fortran」フォルダ内の mpi_c_module.for、mpi_f_func.for、mpi_generic.for ファイルをプロジェクトに追加する。

４）最後に MPI_c 関数を使用したいサブルーチン内で、他の宣言を行う前に **use MPI_DEFINE** 宣言を行い、このモジュールを定義する。以上の手順を踏むことにより Fortran 内で MPI_c を使用することが可能となる(図 5-8)。

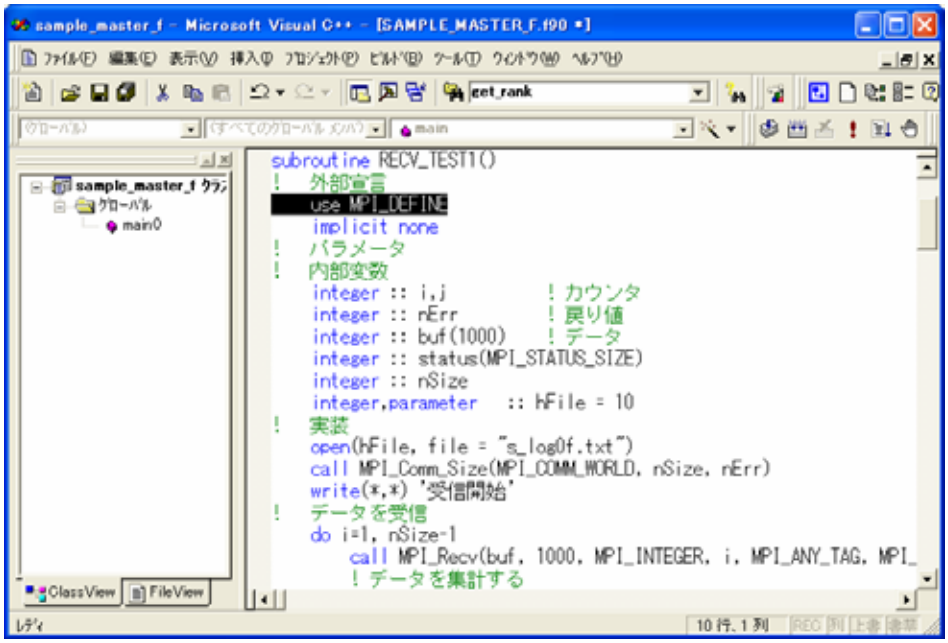

図 5-8　MPI_DEFINE モジュールの使用

5.4 C++用の mpi_c 関数

本節では、C++上で使用できる MPI_c 関数の一覧を表 5-1 に示し、次節以降では、これらの関数ついて具体的に説明する。

表 5-1 MPI_c で使用可能な関数(C++用)

MPI 関数

１）bool **MPI_Init**(int argc, char* argv[])

２）void **MPI_Finalize**()

３）bool **MPI_Comm_size**(MPI_Comm comm = MPI_COMM_WORLD, int *nSize)

４）bool **MPI_Comm_rank**(MPI_Comm comm = MPI_COMM_WORLD, int *nRank)

５） int **MPI_Send**(const void *pBuf, int nCount, MPI_Datatype type, int nDest,int nTag = MPI_ANY_TAG, MPI_Comm comm = MPI_COMM_WORLD)

６） int **MPI_Bcast**(void* pBuf, int nCount, MPI_Datatype type, int nRoot, MPI_Comm comm = MPI_COMM_WORLD)

７） int **MPI_Recv**(void *pBuf, int nCount, MPI_Datatype type, int nSource, int nTag = MPI_ANY_TAG, MPI_Comm comm = MPI_COMM_WORLD, MPI_Status *pStatus = TEMP_STATUS)

８） double **MPI_Wtime**(int num = 0)

拡張 MPI 関数

９） bool **cMPI_Init**(int nID, const char* szParam)

10） int **cMPI_Send**(const void *pBuf, int nSize, int nDest)

11） int **cMPI_Recv**(void *pBuf, int nSize, int nSource)

ユーティリティ関数

12） int **get_size**(int nCount, MPI_Datatype type)

13） int **get_range_s**(int nID)

14） int **get_range_e**(int nID)

15） void **time_log**(const char* szFile)

5.4.1 C++用 MPI 関数

本節では、ライブラリーMPI_c の C++用として実装している関数について説明する。これらの関数は、全て MPI 関数と同じ仕様となっている。

１） 並列初期設定用関数

関数名	bool **MPI_Init**(int argc, char* argv[])
概要	mpi_c の初期化を行う。プロセス設定ファイルの内容を基に、全てのプロセス間接続を行う。
引数　argc	コマンドライン引数の個数
argv	コマンドライン引数
戻り値（型 bool）	正常処理の場合は、true を返す。実行時に引数として渡した値が２個以外であった場合や、初期化に失敗した場合は false を返す。
使用方法	mpi_c 関数を使用する前に、必ず一度この関数をコールしなければならない。 コマンドライン引数をそのまま引数として MPI_Init()に渡せ ばよい。初

	期化の基本的な流れは以下の通りである。 ①プロセス情報の取得 master: プロセス設定ファイルを読み込む。 Slave ： 一つ前のプロセスと接続し、プロセス情報を受け取る。 ②接続アルゴリズム 自分より小さい ID　：接続する。 自分より一つ大きい ID　：起動させ、接続を待ち受ける。 自分より大きい ID　：相手からの接続を待ち受ける。
適用	この関数を使用して、スレーブを起動させる場合は、そのプロセス上の PC では、mpi_c デーモンが動作している必要がある。
例	プロセス設定ファイルの例：このファイルの仕様については第 3.4 節で詳細に説明しているので、そちらを参照されたい。 sf3_slave.exe Dimesnion8200-2　0　230 Dimension8100　231 330 Dimension8100　330 400 Dimension8200-1　401 500

２）並列処理終了関数

関数名	void **MPI_Finalize**(void)
概要	MPI プロセスを全て消去させ、終了処理を行う。
引数	なし
戻り値	なし
使用方法	mpi_c の終了処理を行う。アプリケーションを終了する前に必ず呼び出す必要がある。
適用	

例	// MPI_Init と MPI_Finalize の使用例
	
	#include "mpi_c.h"
	#include <iostream.h>
	int main(int argc, char* argv[])
	{
	　if(MPI_Init(argc, argv) == false)
	　{
	　　cout << "初期化に失敗しました" << endl;
	　　return -1;
	　}
	　MPI_Finalize();
	　return 0;
	}

３）情報取得関数

関数名	bool **MPI_Comm_size**(MPI_Comm comm = MPI_COMM_WORLD, int *nSize)
概要	並列処理プログラムのプロセスでは、自分が誰なのか（自身のランク）を知ることと、他のプロセスがいくつ存在するかを知ることが必要となる。この関数では、自分を含めて全てのプロセス数を知ることができる。
引数　comm	現在は未使用[ダミー]
nSize	プロセス数
戻り値（型 bool）	現在はダミーなので必ず true が返る
使用方法	
適用	
例	

４）情報取得関数

関数名	bool **MPI_Comm_rank**(MPI_Comm comm = MPI_COMM_WORLD, int *nRank)
概要	この関数では、mpi_c 内での自分のランクを取得する。
引数　comm	現在は未使用[ダミー]
nRank	自分のランク
戻り値（型 bool）	現在はダミーなので必ず true が返る
使用方法	
適用	
例	int nSize; int nRank; MPI_Comm_size(MPI_COMM_WORLD, &nSize); cout << "全プロセス数は" << nSize << endl; MPI_Comm_rank(MPI_COMM_WORLD, &nRank; cout << "自分のランクは" << nRank << endl;

５）データ送信用関数

関数名	int **MPI_Send**(const void *pBuf, int nCount, MPI_Datatype type, int nDest,int nTag = MPI_ANY_TAG, MPI_Comm comm = MPI_COMM_WORLD)
概要	指定ランクへのデータの送信を行う。データの送信が完了するかエラーがあるまで操作はブロッキングされる。
引数　pBuf	送信したいデータの先頭アドレス
nCount	送信するデータの個数
type	送信するデータの型(第 5.4.3 節の表 5-3 参照)
nDest	送信先のランク
nTag	タグ番号(現在は未使用) = ID_ANY_TAG

comm	コミュニケータ(現在は未使用) = MPI_COMM_WORLD
戻り値（型 int）	送信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで送信したサイズが返る。
使用方法	
適用	データの送信の順序,型,個数と相手の受信の順序,型,個数が正しく一致しないとデータは正しく転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。

例	int nRet;
	double data[1000];
	// ID=3 にデータを送信する
	nRet = MPI_Send(&data, 1000, MPI_DOUBLE, 3, MPI_ANY_TAG, MPI_COMM_WORLD);
	if(nRet != get_size(1000, MPI_DOUBLE)) cout << "送信失敗" << endl;

６）データ送信用関数

関数名	int **MPI_Bcast**(void* pBuf, int nCount, MPI_Datatype type, int nRoot MPI_Comm comm = MPI_COMM_WORLD)
概要	全てのプロセスにデータを一斉送信する。または一斉送信された値を受信する。
引数　pBuf	送信したいデータの先頭アドレス
nCount	送信するデータの個数
type	送信するデータの型(第 5.4.3 節の表 5-3 参照)
nRoot	送信元のランク
comm	コミュニケータ(現在は未使用) = MPI_COMM_WORLD
戻り値（型 int）	一斉送信を行う側で失敗した場合、nSize 以外の値が返る。 受信する側で失敗した場合は、途中まで受信したサイズが返る。
使用方法	
適用	MPI_Bcast()を用いた場合、データの受信側でも必ず MPI_Bcast()を用いて受信を行わなければならない。MPI_Recv で受信を行った場合の動作は不定であるので注意が必要である。

例	// MPI_Bcast の使用例
	int nRet;
	double data[1000];
	// 一斉送信を行った場合
	nRet = MPI_Bcast(&data, 1000,MPI_DOUBLE, 0, MPI_COMM_WORLD);
	if(nRet != get_size(1000, MPI_DOUBLE))cout << "送受信失敗" << endl;
	
	// MPI_Send,MPI_Recv で代用した場合
	int nRank;
	int nSize;
	int nRet;
	double data[1000];
	MPI_Comm_size(MPI_COMM_WORLD, nSize);
	MPI_Comm_rank(MPI_COMM_WORLD, nRank);
	if(nRank == 0)

	{
	  for(int i=0, i < nSize; i++)
	  {
	  nRet = MPI_Send(&data, 1000, MPI_DOUBLE,0, MPI_ANY_TAG, MPI_COMM_WORLD);
	  }
	}
	else
	{
	  nRet = MPI_Recv(&data, 1000, MPI_DOUBLE,
	  0, MPI_ANY_TAG, MPI_COMM_WORLD,&status);
	}
	if(nRet != get_size(1000, MPI_DOUBLE))cout << "送受信失敗" << endl;

７）データ受信用関数

関数名	int **MPI_Recv**(void *pBuf, int nCount, MPI_Datatype type, int nSourc int nTag = MPI_ANY_TAG, MPI_Comm comm = MPI_COMM_WOR MPI_Status *pStatus = TEMP_STATUS)
概要	指定ランクからデータの受信を行う。データの受信が完了するかエラーがあるまで操作はブロッキングされる。
引数　pBuf	受信したデータの保存先のアドレス
nCount	受信するデータの個数
type	受信するデータの型(第 5.4.3 節の表 5-3 参照)
nSource	送信先のランク
nTag	タグ番号(現在は未使用) = ID_ANY_TAG
comm	コミュニケータ(現在は未使用) = MPI_COMM_WORLD
pStatus	受信ステータスの保存先のアドレス（構造体）
戻り値（型 int）	受信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで受信したサイズが返る。
使用方法	
適用	データの送信の順序,型,個数と相手の受信の順序,型,個数が正しく一致しないとデータは正しく転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。受信ステータスは以下のように構成されている構造体である。 MPI_Status: count = 受信したデータの個数 MPI_SOURCE = 受信元のランク MPI_TAG = 受信したデータのタグ MPI_ERROR = 受信したデータに関するエラー値

例	int nRet;
	double data[1000];
	MPI_STATUS status;
	// ID=0 からデータを受信する
	nRet = MPI_Recv(&data,1000,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD, &status);
	if(nRet != get_size(1000, MPI_DOUBLE))

	cout << "受信失敗" << endl; cout << "個数" << status.count << "ランク" << status.MPI_SOURCE 　　<< "タグ" << status.MPI_TAG << "エラー" << status.MPI_DRROR 　　<< endl;

８）時間計測用関数

関数名	double **MPI_Wtime**(int num = 0)
概要	時間の計測を行う。最初の呼び出しで計測を開始し、二度目の呼び出しで計測を終了し、計測した時間をマイクロ秒単位で関数値として返す。 独立したタイマーを保持しているので MAX_TIMER(mpi_c.h で定義)個を同時に計測することができる。
引数　num	用いるタイマーの番号(0 から MAX_TIMER-1 までの間)
戻り値（型 double）	二度目の呼び出しでは計測された時間(マイクロ秒)を返す。 一度目の呼び出しでは 0,エラーの場合は-1 を返す。
使用方法	
適用	

例	double nTime1, nTime2;
	int nSum;
	nSum = 0;
	nTime1=nTime2=0;
	MPI_Wtime();　　// 計測開始
	for(int i=0; i < 100000; i++)
	{
	　　MPI_Wtime(1);
	　　nSum *= i;
	　　nTime2 = MPI_Wtime(1);
	}
	nTime1 = MPI_Wtime();
	cout << "全体時間:" << nTime1 << "*=:" << nTime2 << endl;

９）C++用拡張 MPI 関数：初期設定関数

関数名	bool **cMPI_Init**(int nID, const char* szParam)
概要	mpi_c の初期化を行う。プロセス設定ファイルの内容を基に、全てのプロセス間接続を行う。
引数　nID	自分の ID
szParam	パラメータ引数 master: プロセス設定ファイルのフルパス slave: 一つ前の ID のコンピュータ名(接続用に用いる)
戻り値（型 bool）	他のプロセスとの接続に失敗した場合は false が返る
使用方法	mpi_c 関数を使用する前に、必ず一度この関数をコールしなければならない。初期化の基本的な流れは以下の通りである。 ①プロセス情報の取得

	master: プロセス設定ファイルを読み込む。 Slave ： 一つ前のプロセスと接続し、プロセス情報を受け取る。 ②接続アルゴリズム 自分より小さい ID　　：接続する。 自分より一つ大きい ID　：起動させ、接続を待ち受ける。 自分より大きい ID　　：相手からの接続を待ち受ける。
適用	この関数を使用して、スレーブを起動させる場合は、そのプロセス上の PC では、mpi_c デーモンが動作している必要がある。

例	#include "mpi_c.h"
	#include <iostream.h>
	int main()
	{
	char szPass[] = "C:¥¥test.txt";
	if(cMPI_Init(0, szPass) == false)
	{
	cout << "初期化に失敗しました" << endl;
	return -1;
	}
	MPI_Finalize();
	return 0;
	}

１０）C++用拡張 MPI 関数：データ送信用関数

関数名	int **cMPI_Send**(const void *pBuf, int nSize, int nDest)
概要	指定ランクへのデータの送信を行う。データの送信が完了するかエラーがあるまで操作はブロッキングされる。
引数　pBuf	送信したいデータの先頭アドレス
nSize	送信したいデータのサイズ(sizeof 演算子で取得)
nDest	送信先のランク
戻り値（型 int）	送信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで送信したサイズが返る。
使用方法	データの送信の順序,データサイズと相手の受信の順序,データサイズが正しく一致しないとデータは正しく転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。
適用	

例	int nRet;
	double data[1000];
	// ID=3 にデータを送信する
	nRet = cMPI_Send(&data, sizeof(data), 3);
	if(nRet != sizeof(data))
	cout << "送信失敗" << endl;

１１）C++用拡張 MPI 関数：データ受信用関数

関数名	int **cMPI_Recv**(void *pBuf, int nSize, int nSource)
概要	指定ランクからデータの受信を行う。データの受信が完了するかエラー

	があるまで操作はブロッキングされる。
引数　pBuf	受信したデータの保存先のアドレス
nSize	受信したいデータのサイズ(sizeof 演算子で取得)
nSource	送信先のランク
戻り値（型 int）	受信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで受信したサイズが返る。
使用方法	
適用	データの送信の順序,データサイズと相手の受信の順序,データサイズが正しく一致しないとデータは正しく転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。

例	int nRet;
	double data[1000];
	// ID=0 からデータを受信する
	nRet = cMPI_Recv(&data, sizeof(data), 0);
	if(nRet != sizeof(data))
	cout << "受信失敗" << endl;

１２）C++用拡張 MPI 関数：データ受信用関数

関数名	int **get_size**(int nCount, MPI_Datatype type)
概要	MPI データ型と個数から必要とするバッファサイズを取得する。
引数　nCount	データの個数
type	MPI データ型(第 5.4.3 節の表 5-3 参照)
戻り値(型 int)	必要とするバッファサイズを返す。 無効なデータ型の場合は-1 を返す
使用方法	
適用	
例	int nSize; nSize = get_size(1000, MPI_DOUBLE); cout << "MPI_DOUBLE-1000 のデータサイズは" << nSize << endl;

１３）C++用拡張 MPI 関数：情報取得用関数

関数名	int **get_range_s**(int nID)
概要	設定ファイルで指定した担当範囲(開始)を取得する。
引数　nID	担当範囲を取得したいプロセスの ID
戻り値(型 int)	担当範囲(開始)を返す
使用方法	
適用	

例	int nStart, nEnd;
	int nProcess;
	MPI_Comm_size(MPI_COMM_WORLD, &nProcess);
	for(int i=0; i < nProcess; i++)
	{
	　　cout << "担当範囲[" << get_range_s(i) << "] - ["
	　　　　<< get_range_e(i) << "]" << endl;
	}

１４）C++用拡張 MPI 関数：情報取得用関数

関数名	int **get_range_e**(int nID)
概要	設定ファイルで指定した担当範囲(終了)を取得する。
引数　nID	担当範囲を取得したいプロセスの ID
戻り値(型 int)	なし: 担当範囲(終了)を返す
使用方法	
適用	
例	

１５）C++用拡張 MPI 関数：情報取得用関数

関数名	bool **time_log**(const char* szFile)
概要	指定されたファイル名に CPU カウンタ周波数と各タイマーのカウンタ値を記録する。
引数　szFile	記録するファイル名
戻り値(型 bool)	ファイルのオープンに失敗すれば false を返す。
使用方法	
適用	

例	// 計測ログの書き出し
	if(time_log("C:¥¥time_log.txt") == false)
	{
	　　cout << "ファイルのオープンに失敗しました" << endl;
	}

5.4.2 Fortran 用 MPI 関数

本節では、Fortran 用 MPI_c の関数について説明する。以下に Fortran 上で使用できる MPI_c 関数の一覧を表 5-2 に示す。

表 5-2　mpi_c で使用可能な関数(Fortran 用)

```
MPI_Init(nErr)
MPI_Finalize()
MPI_Comm_size(comm, nSize, nErr)
MPI_Comm_rank(comm, nRank, nErr)
MPI_Send(buf, nCount, nType, nDest, nTag, nComm, nErr)
MPI_Bcast(buf, nCount, nType, nRoot, nComm, nErr)
MPI_Recv(buf, nCount, nType, nDest, nTag, nComm, status, nErr)
MPI_WTime(num, nTime)
GetRange(nCode, nRank, nRange)
integer*4 GetSize(nCount, nType)
```

上記の関数について詳細に説明する。

１）並列初期設定用関数

関数名	Subroutine **MPI_Init**(nErr)
概要	mpi_c の初期化を行う。プロセス設定ファイルの内容を基に、全てのプロセス間接続を行う。
引数　nErr	エラー値
使用方法	mpi_c 関数を使用する前に、必ず一度この関数をコールしなければならない。 初期化の基本的な流れは以下の通りである。 ①プロセス情報の取得 　master: 設定ファイルを読み込む。 　Slave : 一つ前のプロセスと接続し、プロセス情報を受け取る。 ②接続アルゴリズム 　自分より小さい ID: 接続する。 　自分より一つ大きい ID: 起動させ、接続を待ち受ける。 　自分より大きい ID: 相手からの接続を待ち受ける。
適用	この関数を使用して、スレーブを起動させる場合は、そのプロセスにおける PC 上では、mpi_c デーモンが動作している必要がある。
例	

２）並列処理終了関数

関数名	Subroutine **MPI_Finalize**()
概要	MPI プロセスを全て消去させ、終了処理を行う。

引数	なし
使用方法	mpi_c の終了処理を行う。アプリケーションを終了する前に必ず呼び出す必要がある。
適用	

例	use MPI_DEFINE
	integer nErr
	call MPI_Init(nErr)
	if(nErr .eq. 0) then
	　　write(*,*) "初期化に失敗しました"
	　　return
	end if
	call MPI_Finalize()
	stop
	end

３）情報取得関数

関数名	Subroutine **MPI_Comm_size**(comm, nSize, nErr)
概要	並列処理プログラムのプロセスでは、自分が誰なのか（自身のランク）を知ることと、他のプロセスがいくつ存在するかを知ることが必要となる。この関数では、自分を含めて全てのプロセス数を知ることができる。
引数　comm	integer(IN) コミュニケータ
nSize	integer(OUT) プロセス数
nErr	integer(OUT) エラー値(現在はダミーなので常に１が返る)
使用方法	
適用	
例	use MPI_DEFINE integer nErr integer nProcess call MPI_Comm_size(MPI_COMM_WORLD, nProcess, nErr) write(*,*) "プロセス数=", nProcess

４）情報取得関数

関数名	Subroutine **MPI_Comm_rank**(comm, nRank, nErr)
概要	この関数では、mpi_c 内での自分のランクを取得する。
引数　comm	integer(IN) コミュニケータ
nRank	integer(OUT) 自分のランク
nErr	integer(OUT) エラー値(現在はダミーなので常に１が返る)
使用方法	
適用	
例	use MPI_DEFINE integer nErr integer nRank

	call MPI_Comm_rank (MPI_COMM_WORLD, nRank, nErr) write(*,*) "myRank=", nRank

５）データ送信用関数

関数名	Subroutine **MPI_Send**(pBuf, nCount, nType, nDest, nTag, nComm, nErr)
概要	指定ランクへのデータの送信を行う。データの送信が完了するかエラーがあるまで操作はブロッキングされる。
引数　pBuf	任意の型(IN) 送信したいデータ
nCount	integer(IN) 送信するデータの個数
nType	integer(IN) 送信するデータの型(第 5.4.3 節の表 5-3 参照)
nDest	integer(IN)= 送信先のランク
nTag	integer(INT) タグ番号(現在は未使用)
nComm	integer(IN) コミュニケータ(現在は未使用)
nErr	integer(OUT) エラー値。送信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで送信したサイズが返る
使用方法	
適用	データの送信の順序,型,個数と相手の受信の順序,型,個数が正しく一致しないとデータは正しく転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。

例	use MPI_DEFINE
	integer nErr
	DOUBLE PRECISION data(1000)
	! ランク 3 にデータを送信する
	call MPI_Send(data, 1000, MPI_DOUBLE, 3, MPI_ANY_TAG, MPI_COMM_WORLD, nErr)
	if(nErr .ne. GetSize(1000, MPI_DOUBLE)) then
	　　write(*,*) "送信失敗"
	end if

６）データ送信用関数

関数名	Subroutine **MPI_Bcast**(pBuf, nCount, nType, nRoot, nComm, nErr)
概要	全てのプロセスにデータを一斉送信する。または一斉送信された値を受信する。
引数　pBuf	任意(IN) 送信したいデータ
nCount	integer(IN) 送信するデータの個数
nType	integer(IN) 送信するデータの型(第 5.4.3 節の表 5-3 参照)
nRoot	integer(IN) 送信元のランク
nComm	integer(IN) コミュニケータ(現在は未使用)
nComm	integer(OUT) エラー値。一斉送信を行う側で失敗した場合、nSize 以外の値が返る。受信する側で失敗した場合は、途中まで受信したサイズが

	返る。
使用方法	
適用	MPI_Bcast()を用いた場合、データの受信側でも必ず MPI_Bcast()を用いることにより受信を行わなければならない。MPI_Recv で受信を試みた場合の動作は不定であるので注意が必要である。

例	use MPI_DEFINE
	integer nErr
	DOUBLE PRECISION data(1000)
	! ランク 3 から一斉送信し、他のランクのプロセスはデータを受信する
	call MPI_Bcast(data, 1000, MPI_DOUBLE, 3, MPI_COMM_WORLD, nErr)
	if(nErr .ne. GetSize(1000, MPI_DOUBLE)) then
	　　write(*,*) "一斉送信失敗"
	end if

７）データ受信用関数

関数名	Subroutine **MPI_Recv** pBuf, nCount, nType, nSource, nTag, nComm, pStatus, nErr)
概要	指定ランクからデータの受信を行う。データの受信が完了するかエラーがあるまで操作はブロッキングされる。
引数　pBuf	任意(OUT) 受信したデータの保存先
nCount	integer(IN) 受信するデータの個数
nType	integer(IN) 受信するデータの型(第 5.4.3 節の表 5-3 参照)
nSource	integer(IN) 送信先のランク
nTag	integer(IN) タグ番号(現在は未使用)
comm	integer(IN) コミュニケータ(現在は未使用)
pStatus (MPI_STATUS_SIZE)	Integer(OUT) 受信ステータスを格納する配列
nErr	integer(OUT) = エラー値。受信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで受信したサイズが返る
使用方法	
適用	データの送信の順序,型,個数と相手の受信の順序,型,個数が正しく一致しないとデータは正しく転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。受信ステータスは以下のように構成されている構造体である。 受信ステータスは以下のように構成されている配列である。 pStatus(MPI_STATUS_COUNT)　受信したデータの個数 pStatus(MPI_STATUS_SOURCE)　受信元のランク pStatus(MPI_STATUS_TAG)　受信したデータのタグ pStatus(MPI_STATUS_ERROR)　受信したデータに関する
例	use MPI_DEFINE integer recv_status(MPI_STATUS_SIZE) integer nErr

	DOUBLE PRECISION data(1000)
	// ランク 3 からデータを受信する
	call MPI_Recv(data,1000,MPI_DOUBLE,3,MPI_ANY_TAG,MPI_COMM_WORLD,
	recv_status,nErr)
	if(nErr .ne. GetSize(1000, MPI_DOUBLE)) then
	write(*,*) "受信失敗"
	end if

８）時間計測用関数

関数名	Subroutine **MPI_Wtime**(num, nTime)
概要	時間の計測を行う。最初の呼び出しで計測を開始し、二度目の呼び出しで計測を終了し、計測した時間をマイクロ秒単位で返す。 独立したタイマーを保持しているので MAX_TIMER(mpi_c.h で定義)個を同時に計測することができる。
引数　num	integer(IN) タイマーの番号(0 から MAX_TIMER-1 までの間)
nTime	real*8(OUT) 二度目の呼び出しでは計測された時間(マイクロ秒)を返す。一度目の呼び出しでは 0,エラーの場合は-1 を返す
使用方法	
適用	
例	use MPI_DEFINE real*8 nTime1, nTime2, nStart integer nSum integer i nTime1 = 0 nTime2 = 0 nSum = 0 ! 計測開始 call MPI_Wtime(0, nStart) do i=0, 100000 call MPI_Wtime(1, nStart) nSum = nSum * i call MPI_Wtime(1, nTime2) end do ! 計測終了 call MPI_Wtime(0, nTime1) write(*,*) "全体時間=", nTime1, " * =", nTime2

９）情報取得用関数

関数名	Subroutine **GetRange**(nCode, nRank, nRange)
概要	設定ファイルで指定した担当範囲を取得する。
引数　nCode	integer(IN) 制御コード(1:開始範囲, 2:終了範囲)
nRank	integer(IN) 担当範囲を取得したいプロセスのランク
nRange	integer(OUT) 担当範囲(開始 or 終了)の保存先
使用方法	mpi_c の終了処理を行う。アプリケーションを終了する前に必ず呼び出す必要がある。

適用	
例	use MPI_DEFINE integer nStart, nEnd integer nProcess integer i call MPI_Comm_size(MPI_COMM_WORLD, nProcess, nErr) do i=0, nProcess 　　call GetRange(1, i, nStart) 　　call GetRange(2, i, nEnd) 　　write(*,*) "担当範囲", nStart, "] - [", nEnd, "]" end do

１０）情報取得用関数

関数名	integer*4 Function **GetSize**(nCount, nType)
概要	MPI データ型と個数から必要とするバッファサイズを取得する。
引数　nCount	integer(IN) データの個数
nType	integer(IN) 受信するデータの型(第 5.4.3 節の表 5-3 参照)
戻り値	integer*4　必要とするバッファサイズを返す。無効なデータ型の場合は -1 を返す
使用方法	
適用	
例	use MPI_DEFINE integer nSize nSize = GetSize(1000, MPI_DOUBLE) write(*,*) "MPI_DOUBLE(1000)のサイズ:", nSize

5.4.3 MPI_c 内の定数・構造体

本節では、MPI_c 内で定義されている定数、構造体について説明する。アプリケーションから、つまり MPI_c ライブラリーの外部からも用いられる定数の一覧を表 5-3 に示し、また、内部でのみ用いられる定数の一覧を表 5-4 に示す。外部からも用いられる定数・構造体は、関数の定義と共にヘッダーファイル mpi_c.h に収められており、内部定数の定義は、同じくヘッダーファイル mpi_c_define.h に収められている。

表 5-3　mpi_c 内の各種定義

```
// データタイプの定義
typedef int MPI_Datatype;

#define MPI_CHAR           ((MPI_Datatype)1)  // 文字型
#define MPI_UNSIGNED_CHAR  ((MPI_Datatype)2)  // 符号なし文字型
#define MPI_BYTE           ((MPI_Datatype)3)  // バイト型
#define MPI_SHORT          ((MPI_Datatype)4)  // 単精度整数型
#define MPI_UNSIGNED_SHORT ((MPI_Datatype)5)  // 符号なし単精度整数型
#define MPI_INT            ((MPI_Datatype)6)  // 単精度(16bit)/倍精度(32bit)整数型
#define MPI_UNSIGNED       ((MPI_Datatype)7)  // 符号なし単精度(16bit)/倍精度(32bit)整数型
#define MPI_LONG           ((MPI_Datatype)8)  // 倍精度整数型
#define MPI_UNSIGNED_LONG  ((MPI_Datatype)9)  // 符号なし倍精度整数型
#define MPI_FLOAT          ((MPI_Datatype)10) // 単精度浮動小数点型
#define MPI_DOUBLE         ((MPI_Datatype)11) // 符号なし倍精度浮動小数点型
#define MPI_LONG_DOUBLE    ((MPI_Datatype)12) // 単精度浮動小数点型
#define MPI_LONG_LONG_INT  ((MPI_Datatype)13) // 符号なし倍精度浮動小数点型

// 送信元の定義
#define ID_MASTER  0                  // マスター
#define ID_SLAVE   1                  // スレーブのベースランク
#define MPI_ANY_SOURCE    -2          // ランクのワイルドカード:MPI_ANY_SOURCE

// タグの定義
#define MPI_ANY_TAG       -1          // タグ番号のワイルドカード:MPI_ANY_TAG

// コミュニケータ
typedef int MPI_Comm;                 // コミュニケータ(現在はダミー)
#define MPI_COMM_WORLD 91             // 基本コミュニケータ：MPI_COMM_WORLD

// デフォルト受信ステータスの定義
extern MPI_Status system_stat;
#define TEMP_STATUS system_stat       // 一時使用受信ステータス:TEMP_STATUS

// 最大並列数
#define MAX_PROCESS 64                // 最大プロセス数

// タイマー数の定義
#define MAX_TIMER 64                  // タイマーの数

#define MAX_COMPUTERNAME_LENGTH 16    // ローカルコンピュータ名の最大長さ
```

表 5-4　構造体・内部定数の定義

// 構造体の定義

単精度複素数型 COMPLEX(COMPLEX_4)

```
typedef struct {
      float real;               // 実数部
      float imag;               // 虚数部
} COMPLEX, COMPLEX_4;
```

倍精度複素数型 COMPLEX_8

```
typedef struct {
      double real;        // 実数部
      double imag;        // 虚数部
} COMPLEX_8;
```

受信ステータス MPI_Status

```
typedef struct {
      int count;                // 受信したデータの個数
      int MPI_SOURCE;           // 受信元のランク
      int MPI_TAG;        // 受信したデータのタグ番号
      int MPI_ERROR;            // 受信の際のエラー値
} MPI_Status;
```

プロセス情報 PROCESS_INFO

```
typedef struct {
      // ローカルホスト名
      char  m_szHost[MAX_COMPUTERNAME_LENGTH + 1 ];
      // 相手の ID 値(ID_MASTER = 0, ID_SLAVE + n = 1+n (n=0,1,2,..))
      int         m_nID;
      // 担当範囲[開始 - 終了]
      int         m_nRange[2];
} PROCESS_INFO;
```

内部定数の定義

```
// ポートの定義
#define           PORT_DEAMON        1950   // デーモンポート
#define           PORT_PARALLEL      2000   // 基本ポート
// メッセージ
#define           MPI_MSG_BOOT       1000   // デーモンへのプロセス起動メッセージ
#define           PARAM_SIZE         50     // プロセスを起動する際のコマンドライン
                                                引数の各パラメータの最大文字数
```

5.4.4 MPI_c と MPI_C_Core とのインターフェイス

本節では、実際にソケット関数を用いて通信を実行する MPI_C_Core クラスと MPI_c ライブラリーとのインターフェイスについて説明する。ここで説明する関数は、MPI の仕様に従って定義された関数であり、実際の MPI_C_Core クラスのメンバー関数の仕様に合わせて、関数の引数等を変換する。したがって、関数そのものは非常に単純で理解し易い。最初に、C++用の MPI_c 関数について以下に示す。

```
// MPI 関数の実装
#include "stdafx.h"
#include "mpi_c.h"
#include "MPI_C_Core.h"

#include <stdlib.h>
// システムの実体
CMPI_C_Core core;
// 一時受信ステータスの実体
MPI_Status system_stat;
//    ――――――――――――――――――――――――――――――――
//     ●   MPI 関数群
//    ――――――――――――――――――――――――――――――――
// ================================================================
// MPI 関数群
// DLL の外部から使用できる関数が収められている
// ================================================================

//    ――――――――――――――――――――――――――――――――
//     ●   MPI_Init
//    ――――――――――――――――――――――――――――――――
// ================================================================
// grobal::MPI_Init
// 概要  : mpi_c の初期化を行う。プロセス情報を元に全てのプロセスを接続を行う。
//                 使用方法としては、
//                 コマンドライン引数をそのまま引数として MPI_Init()に渡せばよい。
//                 初期化の基本的な流れは以下の通りである。
//                 ①プロセス情報の取得
//                       master : 設定ファイルを読み込む
//                       slave  : 一つ前のプロセスと接続し、プロセス情報を受け取る。
//                 ②接続アルゴリズム
//                       自分より小さい ID          : 接続する
//                       自分より一つ大きい ID      : 起動させ、接続を待ち受ける
//                       自分より大きい ID          : 相手からの接続を待ち受ける
//          ※  起動させるプロセスでは mpi_c デーモンが起動している必要がある。
// 引数:   argc     = コマンドライン引数の個数
//         argv     = コマンドライン引数
// 戻り値: bool     : 実行時に引数として渡した値が２個以外であった場合や、
//                                 初期化に失敗した場合は false が返る
// ================================================================
bool MPI_Init(int argc, char* argv[])
{
      int nID;
      if(argc != 3)      return false;
```

```
        // 引数を取得        ※  argv[0]は実行ファイル名なので無視
        nID = atoi(argv[1]);       // ID を取得
        return core.Init(nID, argv[2]);
}

//    ―――――――――――――――――――――――――――――――――
//     ●   cMPI_Init
//    ―――――――――――――――――――――――――――――――――
// ================================================================
// grobal::cMPI_Init
// 概要: mpi_c の初期化を行う。プロセス情報を元に全てのプロセスを接続を行う。
//                初期化の基本的な流れは以下の通りである。
//                ①プロセス情報の取得
//                      master : 設定ファイルを読み込む
//                      slave  : 一つ前のプロセスと接続し、プロセス情報を受け取る。
//                ②接続アルゴリズム
//                      自分より小さい ID          : 接続する
//                      自分より一つ大きい ID      : 起動させ、接続を待ち受ける
//                      自分より大きい ID          : 相手からの接続を待ち受ける
//          ※  起動させるプロセスでは mpi_c デーモンが起動している必要がある。
// 引数: nID       = 自分の ID
//       szParam   = パラメータ引数
//                      master: プロセス設定ファイルのフルパス,
//                      slave: 一つ前の ID のコンピュータ名(接続用に用いる)
// 戻り値: bool    : 他のプロセスとの接続に失敗した場合は false が返る
// ================================================================
bool cMPI_Init(int nID, const char* szParam)
{
        return core.Init(nID, szParam);
}

//    ―――――――――――――――――――――――――――――――――
//     ●   MPI_Comm_size
//    ―――――――――――――――――――――――――――――――――
// ================================================================
// grobal::MPI_Comm_size
// 概要: mpi_c で起動しているプロセスの数を取得する
// 引数: MPI_Comm  = 現在は未使用[ダミー]
//                nSize        = プロセス数の保存先
// 戻り値: bool    : 現在はダミーなので必ず true が返る
// ================================================================
bool MPI_Comm_size(MPI_Comm comm, int *nSize)
{
        *nSize = core.GetProcessCount();
        return true;
}

//    ―――――――――――――――――――――――――――――――――
//     ●   MPI_Comm_rank
//    ―――――――――――――――――――――――――――――――――
// ================================================================
// grobal::MPI_Comm_rank
// 概要: mpi_c 内での自分のランクを取得する
// 引数: MPI_Comm   = 現在は未使用[ダミー]
```

```
//      nRank       = 自分のランクの保存先
// 戻り値: bool     : 現在はダミーなので必ず true が返る
// ===================================================================
bool MPI_Comm_rank(MPI_Comm comm, int *nRank)
{
      *nRank = core.GetMyID();
      return true;
}

//    ―――――――――――――――――――――――――――――――――
//     ●   MPI_Send
//    ―――――――――――――――――――――――――――――――――
// ===================================================================
// grobal::MPI_Send
// 概要: 指定ランクへのデータの送信を行う。
//                 データの送信が完了するかエラーがあるまで操作はブロッキングされる。
//           ※   データの送信の順序,型,個数と相手の受信の順序,型,個数が正しく一致
//                 しないとデータは正しく転送されず、アプリケーションはデッドロックを
//                 起こし、止まってしまうので細心の注意が必要である。
// 引数: pBuf      = 送信したいデータの先頭アドレス
//                 nCount = 送信するデータの個数
//                 type   = 送信するデータの型(mpi_c.h で指定されているデータ型)
//                 nDest  = 送信先のランク
//                 nTag   = タグ番号(現在は未使用)
//                 comm   = コミュニケータ(現在は未使用)
// 戻り値: int     : 送信したデータのサイズが返る。
//                       エラーなどで途中終了した場合は途中まで送信したサイズが返る
// ===================================================================
int MPI_Send(const void *pBuf, int nCount, MPI_Datatype type, int nDest, int nTag, MPI_Comm comm)
{
      return core.Send(nDest, pBuf, get_size(nCount, type));
}

//    ―――――――――――――――――――――――――――――――――
//     ●   cMPI_Send
//    ―――――――――――――――――――――――――――――――――
// ===================================================================
// grobal::cMPI_Send
// 概要: 指定ランクへのデータの送信を行う。
//                 データの送信が完了するかエラーがあるまで操作はブロッキングされる。
//           ※   データの送信の順序,データサイズと相手の受信の順序,データサイズが
//                 正しく一致しないとデータは正しく転送されず、アプリケーションは
//                 デッドロックを起こし、止まってしまうので細心の注意が必要である。
// 引数: pBuf      = 送信したいデータの先頭アドレス
//                 nSize  = 送信したいデータのサイズ(sizeof 演算子で取得)
//                 nDest  = 送信先のランク
// 戻り値: int   : 送信したデータのサイズが返る。
//                 エラーなどで途中終了した場合は途中まで送信したサイズが返る
// ===================================================================
int cMPI_Send(const void *pBuf, int nSize, int nDest)
{
      return core.Send(nDest, pBuf, nSize);
}
//    ―――――――――――――――――――――――――――――――――
```

```
//     ●   MPI_Bcast
//    ―――――――――――――――――――――――――――――――――――――
// ====================================================================
// grobal::MPI_Bcast
// 概要: 全てのプロセスにデータを一斉送信する。
//          ※ MPI_Bcast()を用いた場合、データの受信側でも必ず
//                MPI_Bcast()を用いることにより受信を行わなければならない。
//                MPI_Recv で受信を試みた場合の動作は不定であるので
//                注意が必要である。
// 引数: pBuf      = 送信データの先頭アドレス(データ型は問わない)
//                nCount = 送信するデータの個数
//                type   = 送信するデータの型(mpi_c.h で指定されているデータ型)
//                nRoot  = 送信元プロセスの ID
//                comm   = コミュニケータ(現在は未使用)
// 戻り値: int : 一斉送信を行う側で失敗した場合、nSize 以外の値が返る
//              受信する側で失敗した場合は、途中まで受信したサイズが返る
// ====================================================================
int MPI_Bcast(void* pBuf, int nCount, MPI_Datatype type, int nRoot, MPI_Comm comm)
{
      return core.BCast(nRoot, pBuf, get_size(nCount, type));
}

//    ―――――――――――――――――――――――――――――――――――――
//     ●   MPI_Recv
//    ―――――――――――――――――――――――――――――――――――――
// ====================================================================
// grobal::MPI_Recv
// 概要: 指定ランクからデータの受信を行う。
//                データの受信が完了するかエラーがあるまで操作はブロッキングされる。
//          ※  データの送信の順序,サイズと相手の受信の順序,サイズが正しく一致
//                しないとデータは正しく転送されず、アプリケーションはデッドロックを
//                起こし、止まってしまうので細心の注意が必要である。
//                受信ステータスは以下のように構成されている構造体である。
//          MPI_Status:  count        = 受信したデータの個数
//                           MPI_SOURCE   = 受信元のランク
//                           MPI_TAG            = 受信したデータのタグ
//                           MPI_ERROR    = 受信したデータに関するエラー値
// 引数: pBuf      = 受信したデータの保存先のアドレス
//                nCount = 受信するデータの個数
//                type   = 受信するデータの型(mpi_c.h で指定されているデータ型)
//                nSource      = 受信元のランク
//                nTag   = タグ番号(現在は未使用)
//                comm   = コミュニケータ(現在は未使用)
//                pStatus      = 受信ステータスの保存先
// 戻り値: int     : 受信したデータのサイズが返る。
//                      エラーなどで途中終了した場合は途中まで受信したサイズが返る
// ====================================================================
int MPI_Recv(void *pBuf, int nCount, MPI_Datatype type, int nSource, int nTag,
            MPI_Comm comm, MPI_Status *pStatus)
{
      int i;
      int nRet;
      int nProc = core.GetProcessCount();
      // ランクの指定がない場合
```

```
        if(nSource == MPI_ANY_SOURCE)
        {
              // select に渡すソケット構造体をセット
              fd_set fds;
              FD_ZERO(&fds);
              for(i=0; i < nProc; i++)
              {
              FD_SET(core.m_pSocket[i], &fds);
              }
              // 受信可能になるまで待ち受け
              nRet = select(fds.fd_count+1, &fds, NULL, NULL, NULL);
              // 受信可能になったソケットを探す
              while(nSource == MPI_ANY_SOURCE)
              {
              for(i=0; i < nProc; i++)
               {
                nRet = sock_recv_check(core.m_pSocket[i]);
                if(nRet > 0){
                 nSource = i;           // 受信可能なソケットを指定ランクとする
                 break;
                }
              }
              }
        }
        // 指定ランクから受信する
        nRet = core.Recv(nSource, pBuf, get_size(nCount, type));
        // 受信ステータスをセット
        pStatus->count           = nCount;
        pStatus->MPI_SOURCE      = nSource;
        pStatus->MPI_TAG         = nTag;
        pStatus->MPI_ERROR       = nRet;
        return nRet;
  }

  //    ―――――――――――――――――――――――――――――――
  //     ●   cMPI_Recv
  //    ―――――――――――――――――――――――――――――――
  // ================================================================
  // grobal::cMPI_Recv
  // 概要: 指定ランクからデータの受信を行う。
  //                データの受信が完了するかエラーがあるまで操作はブロッキングされる。
  //          ※  データの送信の順序,サイズと相手の受信の順序,サイズが正しく一致
  //                しないとデータは正しく転送されず、アプリケーションはデッドロックを
  //                起こし、止まってしまうので細心の注意が必要である。
  // 引数: pBuf      = 受信したデータの保存先のアドレス
  //                nSize  = 受信したいデータのサイズ(sizeof 演算子で取得)
  //                nSource      = 受信元のランク
  // 戻り値: int     : 受信したデータのサイズが返る。
  //                 エラーなどで途中終了した場合は途中まで受信したサイズが返る
  // ================================================================
  int cMPI_Recv(void *pBuf, int nSize, int nSource)
  {
        return core.Recv(nSource, pBuf, nSize);
  }
```

```
//    ―――――――――――――――――――――――――――――――――
//      ●   MPI_Finalize
//    ―――――――――――――――――――――――――――――――――
// ================================================================
// grobal::MPI_Finalize
// 概要   : mpi_c の終了処理を行う。アプリケーションを終了する前に必ず呼び出す
//                必要がある。
// 引数   : none
// 戻り値: none
// ================================================================
void MPI_Finalize()
{
      core.ShutDownAll();
}

//    ―――――――――――――――――――――――――――――――――
//      ●   MPI_Wtime
//    ―――――――――――――――――――――――――――――――――
// ================================================================
// grobal::MPI_Wtime
// 概要   : 時間の計測を行う。最初の呼び出しで計測を開始し、
//                二度目の呼び出しで計測を終了し、計測した時間をマイクロ秒単位で返す。
//                独立したタイマーを保持しているので
//                MAX_TIMER(mpi_c.h で定義)個を同時に計測することができる。
// 引数   : num      = 用いるタイマーの番号(0 から MAX_TIMER-1 までの間)
// 戻り値: double   : 二度目の呼び出しでは計測された時間(マイクロ秒)を返す。
//                  一度目の呼び出しでは 0,エラーの場合は-1 を返す
// ================================================================
double MPI_Wtime(int num)
{
      double nRet;
      static bool s_bStat[MAX_TIMER] = {false};    // 開始／終了フラグ
      s_bStat[num] = s_bStat[num] ? false : true;  // 開始／終了フラグの ON・OFF を切り替える

      if(s_bStat[num] == true)
      {
            nRet = core.Time_Start(num);
      }
      else
      {
            nRet = core.Time_Stop(num);
      }
      return nRet;
}

//    ―――――――――――――――――――――――――――――――――
//      ●   get_size
//    ―――――――――――――――――――――――――――――――――
// ================================================================
// grobal::get_size
// 概要   : MPI データ型と個数から必要とするバッファサイズを取得する。
// 引数   :   nCount = データの個数
//            type   = MPI データ型(mpi_c.h で指定されているデータ型)
```

```
// 戻り値: int      : 必要とするバッファサイズを返す。
//                            無効なデータ型の場合は-1 を返す
// ================================================================
int get_size(int nCount, MPI_Datatype type)
{
      if(nCount <= 0)    return -1;
      switch(type)
      {
      case MPI_CHAR:
            return sizeof(char) * nCount;
      case MPI_UNSIGNED_CHAR:   case MPI_CHARACTER:
            return sizeof(unsigned char) * nCount;
      case MPI_BYTE:
            return sizeof(unsigned char) * nCount;
      case MPI_SHORT:
            return sizeof(short) * nCount;
      case MPI_UNSIGNED_SHORT:
            return sizeof(unsigned short) * nCount;
      case MPI_INT:      case MPI_LOGICAL: case MPI_INTEGER:
            return sizeof(int) * nCount;
      case MPI_UNSIGNED:
            return sizeof(unsigned) * nCount;
      case MPI_LONG:
            return sizeof(long) * nCount;
      case MPI_UNSIGNED_LONG:
            return sizeof(unsigned long) * nCount;
      case MPI_FLOAT:    case MPI_REAL:
            return sizeof(float) * nCount;
      case MPI_DOUBLE:   case MPI_DOUBLE_PRECISION:
            return sizeof(double) * nCount;
      case MPI_LONG_DOUBLE:
            return sizeof(long double) * nCount;
      case MPI_COMPLEX:
            return sizeof(COMPLEX) * nCount;
      case MPI_DOUBLE_COMPLEX:
            return sizeof(COMPLEX_8) * nCount;
      default:
            return -1;
      }
}

//    ――――――――――――――――――――――――――――――――
//     ●   get_range_s
//    ――――――――――――――――――――――――――――――――
// ================================================================
// grobal::get_range_s
// 概要   : 担当範囲(開始)を取得する
// 引数   : nID      = 担当範囲を取得したいプロセスの ID
// 戻り値: int      : 担当範囲(開始)を返す
// ================================================================
int get_range_s(int nID)
{
      return core.GetRange_S(nID);
}
```

```
// ================================================================
// grobal::get_range_e
// 概要　: 担当範囲(終了)を取得する
// 引数　:　 nID　= 担当範囲を取得したいプロセスの ID
// 戻り値: int　　: 担当範囲(終了)を返す
// ================================================================
int get_range_e(int nID)
{
	return core.GetRange_E(nID);
}

//　―――――――――――――――――――――――――――――――
//　 ●　time_log
//　―――――――――――――――――――――――――――――――
// ================================================================
// grobal::time_log
// 概要　: 指定されたファイル名に CPU カウンタ周波数と
//　　　　　各タイマーのカウンタ値を記録する。
// 引数　: szFile　= 記録するファイル名
// 戻り値: bool　　: ファイルのオープンに失敗すれば false を返す
// ================================================================
bool time_log(const char* szFile)
{
	return core.TimeLog(szFile);
}
```

上記の関数の中で、使用されている MPI_C_Core クラスの関数を以下にまとめる。これらについては、後節で説明する。

```
core.Init(nID, argv[2])
core.GetProcessCount()
core.GetMyID()
core.Send()
core.BCast()
core.GetProcessCount()
sock_recv_check()
core.Recv()
core.ShutDownAll()
core.Time_Start(num)
core.Time_Stop(num)
core.GetRange_S(nID)
core.GetRange_E(nID)
core.TimeLog(szFile)
core.TimeLog(szFile)
```

5.5 Fortran 用の MPI_c 関数

本節では、Fortran プログラムより MPI_c ライブラリーを利用できる

関数について説明する。以下に実装した MPI_c に関する Fortran サブルーチンを示す。

```
C     ――――――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE / MPI_Init
C     ――――――――――――――――――――――――――――――――――――
C     ●   MPI の初期化
C     ――――――――――――――――――――――――――――――――――――
      subroutine MPI_Init(nErr)
c     外部宣言
            use DFLIB
            use MPI_C_Func
            implicit none
c     引数宣言
            INTEGER*4    :: nErr
c     内部変数
            integer*4 :: nArgc
            CHARACTER(18) :: szArgv1, szArgv2
            CHARACTER(20) :: szIP
            INTEGER*4    :: nRank
c     実装
            ! コマンドライン引数を取得
            call GetArg(1, szArgv1)
            call GetArg(2, szArgv2)

            read(szArgv1, '(I4)') nRank     ! rank を取得
            write(szIP, '(2A)') szArgv2, ''C! IP に NULL 文字を追加(C++用)
            nErr = cMPI_Init(%VAL(nRank), %REF(szIP))
      end subroutine

C     ――――――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE / MPI_Comm_size
C     ――――――――――――――――――――――――――――――――――――
C     ●   全プロセス数を取得する
C     ――――――――――――――――――――――――――――――――――――
      subroutine MPI_Comm_size(comm, nSize, nErr)
c     外部宣言
            use MPI_C_Func
            implicit none
c     引数宣言
            integer*4 :: comm
            integer*4 :: nSize
            integer*4 :: nErr
c     実装
            nErr = cMPI_Comm_size(%VAL(comm), %REF(nSize))
      end subroutine

C     ――――――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE / MPI_Comm_rank
C     ――――――――――――――――――――――――――――――――――――
C     ●   自分のランクを取得する
C     ――――――――――――――――――――――――――――――――――――
      subroutine MPI_Comm_rank(comm, nRank, nErr)
```

```
c     外部宣言
            use MPI_C_Func
            implicit none
c     宣言
            integer*4 :: comm
            integer*4 :: nRank
            integer*4 :: nErr
c     実装
            nErr = cMPI_Comm_rank(%VAL(comm), %REF(nRank))
      end subroutine

C     ―――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE / GetRange
C     ―――――――――――――――――――――――――――――――――
C     ●   指定ランクの担当範囲を取得する
C     ―――――――――――――――――――――――――――――――――
      subroutine GetRange(nCode, nRank, nRange)
c     外部宣言
            use MPI_C_Func
            implicit none
c     宣言
            integer*4 :: nCode        ! 制御コード(1:開始範囲, 2:終了範囲)
            integer*4 :: nRank        ! 指定ランク
            integer*4 :: nRange       ! 担当範囲
c     実装
            if(nCode == 1) then
                  nRange = cGetRange_s(%VAL(nRank))
            else
                  nRange = cGetRange_e(%VAL(nRank))
            end if
      end subroutine

C     ―――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE / MPI_WTime
C     ―――――――――――――――――――――――――――――――――
C     ●   指定ランクの担当範囲を取得する
c           １回目のコールでスタート、２回目のコールでストップ。計測時間を返す
C     ―――――――――――――――――――――――――――――――――
      subroutine MPI_WTime(num, nTime)
c     外部宣言
            use MPI_C_Func
            implicit none
c     宣言
            integer*4    :: num       ! タイマーの番号(最大:MAX_TIMER)
            real*8       :: nTime           ! 計測時間(マイクロ秒)
c     実装
            nTime = cMPI_WTime(%VAL(num))
      end subroutine
C     ―――――――――――――――――――――――――――――――――
C     ●   SUBROUTINE / GetSize
C     ―――――――――――――――――――――――――――――――――
C     ●   指定した型と個数からデータサイズを取得する
C     ―――――――――――――――――――――――――――――――――
      integer*4 function GetSize(nCount, nType)
```

```
c     外部宣言
            use MPI_C_Func
            implicit none
c     宣言
            integer*4    :: nCount    ! データの個数
            integer*4    :: nType     ! データの型
c     実装
            GetSize = cGetSize(%VAL(nCount), %VAL(nType))
      end function


!     mpi_c の送受信関数の実装

C     ――――――――――――――――――――――――――――――――
c     ●   SUBROUTINE / MPI_Send
C     ――――――――――――――――――――――――――――――――
      SUBROUTINE MPI_Send
     &            (buf, nCount, nType, nDest, nTag, nComm, nErr)
            use MPI_C_FUNC, only : cMPI_Send
            implicit none
c           引数の宣言
            INTEGER      :: buf
            INTEGER      :: nCount
            INTEGER      :: nType
            INTEGER      :: nDest
            INTEGER      :: nTag
            INTEGER      :: nComm
            INTEGER      :: nErr
c           内部変数の宣言
            INTEGER      :: pBuf
c           実装
            pBuf = LOC(buf)
            nErr = cMPI_Send(%VAL(pBuf), %VAL(nCount), %VAL(nType),
     &                               %VAL(nDest), %VAL(nTag), %VAL(nComm))
      END SUBROUTINE

C     ――――――――――――――――――――――――――――――――
c     ●   SUBROUTINE / MPI_Bcast
C     ――――――――――――――――――――――――――――――――
      SUBROUTINE MPI_Bcast
     &            (buf, nCount, nType, nRoot, nComm, nErr)
            use MPI_C_FUNC, only : cMPI_Bcast
            implicit none
c           引数の宣言
            INTEGER      :: buf
            INTEGER      :: nCount
            INTEGER      :: nType
            INTEGER      :: nRoot
            INTEGER      :: nComm
            INTEGER      :: nErr
c           内部変数の宣言
            INTEGER      :: pBuf
c           実装
            pBuf = LOC(buf)
```

```
            nErr = cMPI_Bcast(%VAL(pBuf), %VAL(nCount), %VAL(nType),
     &                                    %VAL(nRoot),  %VAL(nComm))
      END SUBROUTINE

C     ―――――――――――――――――――――――――――――――――
c     ●   SUBROUTINE / MPI_Recv
C     ―――――――――――――――――――――――――――――――――
      SUBROUTINE MPI_Recv
     &            (buf, nCount, nType, nDest, nTag, nComm, status, nErr)
            use MPI_DEFINE, only : MPI_STATUS_SIZE, MPI_STATUS
            use MPI_C_FUNC, only : cMPI_Recv
            implicit none
c           引数の宣言
            INTEGER, INTENT(INOUT)    :: buf
            INTEGER, INTENT(INOUT)    :: nCount
            INTEGER, INTENT(INOUT)    :: nType
            INTEGER, INTENT(INOUT)    :: nDest
            INTEGER, INTENT(INOUT)    :: nTag
            INTEGER, INTENT(INOUT)    :: nComm
            INTEGER, INTENT(INOUT)    :: status(MPI_STATUS_SIZE)
            INTEGER, INTENT(INOUT)    :: nErr
c           内部変数の宣言
            INTEGER :: i
            INTEGER      :: pBuf
            TYPE(MPI_STATUS) :: recv_stat
c           初期化
            recv_stat%nCount   = 0
            recv_stat%nSource  = 0
            recv_stat%nTag     = 0
            recv_stat%nErr     = 0
c           実装
            pBuf = LOC(buf)                                    ! アドレスを格納
            ! 受信
            nErr = cMPI_Recv(%VAL(pBuf), %VAL(nCount), %VAL(nType),
     &            %VAL(nDest), %VAL(nTag), %VAL(nComm), %REF(recv_stat))
            ! 受信ステータスをセット
            status(1)    = recv_stat%nCount
            status(2)    = recv_stat%nSource
            status(3)    = recv_stat%nTag
            status(4)    = recv_stat%nErr
      END SUBROUTINE
```

Fortran のサブルーチンで、MPI_c の関数を用いる場合、module を以下の use 文でインクルードする必要がある。ただし、only :　以下の構造体や関数名は、その前の module の中の該当する構造体や関数にのみ適用することを意味する。

```
      use MPI_DEFINE, only : MPI_STATUS_SIZE, MPI_STATUS
      use MPI_C_FUNC, only : cMPI_Recv
      use MPI_GENERIC_FUNC
```

以下に示す２つの module で、最初の module はパラメータや定数の定

義や、構造体の定義文をまとめたものである。後の module は、MPI_c 関数のインターフェイス文をまとめたもので、C++で書かれたこれらの関数を Fortran プログラムから呼ぶ場合は、この module を必ずインクルードしなければならない。

```
C     ―――――――――――――――――――――――――――――――――――
C     ●   MODULE / MPI_DEFINE
C     ―――――――――――――――――――――――――――――――――――
C     ●   mpi_c 関数を Fortran から用いるための参照モジュール
C     ―――――――――――――――――――――――――――――――――――
      MODULE MPI_DEFINE
      implicit none
      INTEGER*4, PARAMETER ::          ! fortran : C++
     & MPI_COMPLEX  = 23,              ! COMPLEX : struct COMPLEX
     & MPI_DOUBLE_COMPLEX = 24,        ! DOUBLE_COMPLEX  : struct COMPLEX_8
     & MPI_LOGICAL  = 25,              ! LOGICAL = int
     & MPI_REAL     = 26,              ! REAL : double
     & MPI_DOUBLE_PRECISION = 27,      ! DOUBLE_PRECISION   : double
     & MPI_DOUBLE   = 27,
     & MPI_INTEGER  = 28,              ! INT : int
     & MPI_CHARACTER      = 33         ! CHARCTER = unsinged char

!       タグの定義
      INTEGER*4, PARAMETER ::   MPI_ANY_TAG  = -1
!       ID の定義
      INTEGER*4, PARAMETER ::   ID_MASTER    = 0,
     &                          ID_SLAVE     = 1,
     &                  MPI_ANY_SOURCE     = -2  ! 現在は使用不可能
!       コミュニケータの定義
      INTEGER*4, PARAMETER :: MPI_COMM_WORLD = 91
!     MPI_STATUS のサイズ
      INTEGER*4, PARAMETER :: MPI_STATUS_SIZE = 4
!     MPI_STATUS の番号
      INTEGER*4, PARAMETER :: MPI_STATUS_COUNT     = 1
      INTEGER*4, PARAMETER :: MPI_STATUS_SOURCE    = 2
      INTEGER*4, PARAMETER :: MPI_STATUS_TAG       = 3
      INTEGER*4, PARAMETER :: MPI_STATUS_ERROR     = 4

!―――――――――――――――――――――――――――――――――――――――――
!     ※  拡張するときは、以下に記してください。
!―――――――――――――――――――――――――――――――――――――――――

c―――――――――――――――――――――――――――――――★構造体宣言
C     ―――――――――――――――――――――――――――――――――――
C     ●   mystatus 構造体
C     ―――――――――――――――――――――――――――――――――――
            TYPE mystatus
                  integer    Rank                          !  自分の MPI の Rank
                  integer    GroupCount    !  MPI 全体の数
            end TYPE
C     ―――――――――――――――――――――――――――――――――――
C     ●   MPI_STATUS 構造体
```

```
C     ―――――――――――――――――――――――――――――――――――――
        TYPE  MPI_STATUS
              integer nCount;          ! 受信データの個数
              integer nSource;         ! 送信元のランク
              integer nTag;            ! 受信データのタグ
              integer nErr;            ! エラーコード
        end TYPE
      END MODULE
C     ―――――――――――――――――――――――――――――――――――――


C     ―――――――――――――――――――――――――――――――――――――
C     ●   MODULE / MPI_C_Func
C     ―――――――――――――――――――――――――――――――――――――
C     ●   mpi_c 関数を Fortran から用いるための参照モジュール
C     ―――――――――――――――――――――――――――――――――――――
      MODULE MPI_C_FUNC
C     ―――――――――――――――――――――――――――――――――――――
C     ●   cMPI_Init
C     ―――――――――――――――――――――――――――――――――――――
        INTERFACE
          INTEGER*4 FUNCTION cMPI_Init(nID, szIP)
            !DEC$ ATTRIBUTES C, ALIAS:'_cMPI_Init' :: cMPI_Init
            INTEGER*4 :: nID[VALUE]
            CHARACTER*(*) :: szIP[REFERENCE]
          END FUNCTION
        END INTERFACE
C     ―――――――――――――――――――――――――――――――――――――
C     ●   MPI_Finalize
C     ―――――――――――――――――――――――――――――――――――――
        INTERFACE
          SUBROUTINE MPI_Finalize()
            !DEC$ ATTRIBUTES C, ALIAS:'_MPI_Finalize' :: MPI_Finalize
          END SUBROUTINE
        END INTERFACE
C     ―――――――――――――――――――――――――――――――――――――
C     ●   cMPI_Comm_size
C     ―――――――――――――――――――――――――――――――――――――
        INTERFACE
          INTEGER*4 FUNCTION cMPI_Comm_size(comm, nSize)
            !DEC$ ATTRIBUTES C, ALIAS:'_MPI_Comm_size' :: cMPI_Comm_size
            INTEGER*4 :: comm[VALUE]
            INTEGER*4 :: nSize[REFERENCE]
          END FUNCTION
        END INTERFACE
C     ―――――――――――――――――――――――――――――――――――――
C     ●   cMPI_Comm_rank
C     ―――――――――――――――――――――――――――――――――――――
        INTERFACE
          INTEGER*4 FUNCTION cMPI_Comm_rank(comm, nRank)
            !DEC$ ATTRIBUTES C, ALIAS:'_MPI_Comm_rank' :: cMPI_Comm_rank
            INTEGER*4 :: comm[VALUE]
            INTEGER*4 :: nRank[REFERENCE]
          END FUNCTION
```

```
            END INTERFACE
C     ―――――――――――――――――――――――――――――――――――
C     ●   cMPI_Send
C     ―――――――――――――――――――――――――――――――――――
            INTERFACE
                  INTEGER*4 FUNCTION cMPI_Send
     &            (pBuf, nCount, nType, nDest, nTag, comm)
                        !DEC$ ATTRIBUTES C, ALIAS:'_MPI_Send' :: cMPI_Send
                        INTEGER*4 :: pBuf[VALUE]
                        INTEGER*4 :: nCount[VALUE]
                        INTEGER*4 :: nType[VALUE]
                        INTEGER*4 :: nDest[VALUE]
                        INTEGER*4 :: nTag[VALUE]
                        INTEGER*4 :: comm[VALUE]
                  END FUNCTION
            END INTERFACE
C     ―――――――――――――――――――――――――――――――――――
C     ●   cMPI_Bcast
C     ―――――――――――――――――――――――――――――――――――
            INTERFACE
                  INTEGER*4 FUNCTION cMPI_Bcast
     &            (pBuf, nCount, nType, nRoot, comm)
                        !DEC$ ATTRIBUTES C, ALIAS:'_MPI_Bcast' :: cMPI_Bcast
                        INTEGER*4 :: pBuf[VALUE]
                        INTEGER*4 :: nCount[VALUE]
                        INTEGER*4 :: nType[VALUE]
                        INTEGER*4 :: nRoot[VALUE]
                        INTEGER*4 :: comm[VALUE]
                  END FUNCTION
            END INTERFACE
C     ―――――――――――――――――――――――――――――――――――
C     ●   cMPI_Recv
C     ―――――――――――――――――――――――――――――――――――
            INTERFACE
                  INTEGER*4 FUNCTION  cMPI_Recv
     &      (pBuf, nCount, nType, nSource, nTag, comm, pStatus)
                        !DEC$ ATTRIBUTES C, ALIAS:'_MPI_Recv' :: cMPI_Recv
                        INTEGER*4 :: pBuf[VALUE]
                        INTEGER*4 :: nCount[VALUE]
                        INTEGER*4 :: nType[VALUE]
                        INTEGER*4 :: nSource[VALUE]
                        INTEGER*4 :: nTag[VALUE]
                        INTEGER*4 :: comm[VALUE]
                        INTEGER*4 :: pStatus[VALUE]
                  END FUNCTION
            END INTERFACE
C     ―――――――――――――――――――――――――――――――――――
C     ●   cGetRange_s
C     ―――――――――――――――――――――――――――――――――――
            INTERFACE
                  INTEGER*4 FUNCTION cGetRange_s(nID)
                        !DEC$ ATTRIBUTES C, ALIAS:'_get_range_s' :: cGetRange_s
                        INTEGER*4 :: nID[VALUE]
                  END FUNCTION
```

```
            END INTERFACE
C     ―――――――――――――――――――――――――――――――――――
C     ●   cGetRange_e
C     ―――――――――――――――――――――――――――――――――――
            INTERFACE
                  INTEGER*4 FUNCTION cGetRange_e(nID)
                        !DEC$ ATTRIBUTES C, ALIAS:'_get_range_e' :: cGetRange_e
                        INTEGER*4 :: nID[VALUE]
                  END FUNCTION
            END INTERFACE
C     ―――――――――――――――――――――――――――――――――――
C     ●   cMPI_WTime
C     ―――――――――――――――――――――――――――――――――――
            INTERFACE
                  REAL*8 FUNCTION cMPI_WTime(num)
                        !DEC$ ATTRIBUTES C, ALIAS:'_MPI_Wtime' :: cMPI_WTime
                        INTEGER*4 :: num[VALUE]
                  END FUNCTION
            END INTERFACE
C     ―――――――――――――――――――――――――――――――――――
C     ●   cGetSize
C     ―――――――――――――――――――――――――――――――――――
            INTERFACE
                  INTEGER*4 FUNCTION cGetSize(nCount, nType)
                        !DEC$ ATTRIBUTES C, ALIAS:'_get_size' :: cGetSize
                        INTEGER*4 :: nCount[VALUE]
                        INTEGER*4 :: nType[VALUE]
                  END FUNCTION
            END INTERFACE
      END MODULE
C     ―――――――――――――――――――――――――――――――――――

C     ―――――――――――――――――――――――――――――――――――
C     ●   MODULE / MPI_GENERIC_FUNC
C     ―――――――――――――――――――――――――――――――――――
C     ●   void*型を含む C++関数を取り扱うための総称関数モジュール
C     ―――――――――――――――――――――――――――――――――――
      MODULE MPI_GENERIC_FUNC
            INTERFACE
C     ―――――――――――――――――――――――――――――――――――
c     ●   SUBROUTINE / MPI_Send
C     ―――――――――――――――――――――――――――――――――――
            SUBROUTINE MPI_Send
     &      (buf, nCount, nType, nDest, nTag, nComm, nErr)
                  !DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
c                 引数の宣言
                  INTEGER     :: buf
                  INTEGER     :: nCount
                  INTEGER     :: nType
                  INTEGER     :: nDest
                  INTEGER     :: nTag
                  INTEGER     :: nComm
                  INTEGER     :: nErr
            END SUBROUTINE
```

```
C     ―――――――――――――――――――――――――――――――――
c     ●   SUBROUTINE / MPI_Bcast
C     ―――――――――――――――――――――――――――――――――
            SUBROUTINE MPI_Bcast
     &      (buf, nCount, nType, nRoot, nComm, nErr)
                  !DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
c                 引数の宣言
                  INTEGER     :: buf
                  INTEGER     :: nCount
                  INTEGER     :: nType
            INTEGER     :: nRoot
                  INTEGER     :: nComm
                  INTEGER     :: nErr
            END SUBROUTINE

C     ―――――――――――――――――――――――――――――――――
c     ●   SUBROUTINE / MPI_Recv
C     ―――――――――――――――――――――――――――――――――
            SUBROUTINE MPI_Recv
     &            (buf, nCount, nType, nDest, nTag, nComm, status, nErr)
                  !DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
c                 引数の宣言
                  INTEGER     :: buf
                  INTEGER     :: nCount
                  INTEGER     :: nType
                  INTEGER     :: nDest
                  INTEGER     :: nTag
                  INTEGER     :: nComm
                  INTEGER     :: status(5)
                  INTEGER     :: nErr
            END SUBROUTINE
            END INTERFACE
       END MODULE
C     ―――――――――――――――――――――――――――――――――
```

5.6 CMPI_C_Core クラス

MPI_c ライブラリーの本体は、CMPI_C_Core クラスに定義されており、グローバルオブジェクト core としてシステム中に存在している。並列プログラムから MPI_c 関数を呼び出すと、MPI_c 関数は core オブジェクトにアクセスして通信処理を行う。ここでは、CMPI_C_Core の関数・オブジェクトの一覧を表 5-5 に示す。

表 5-5　CMPI_C_Core クラス機能一覧

```
// 初期化・終了
bool Init(int nID, const char* szParam)      // mpi_c の初期化処理
void ShutDownAll()                           // 全ての接続を切断する

// 送受信
inline int Send(int nID, const void* pBuf, int nSize)     // データの送信を行う
inline int Recv(int nID, void* pBuf, int nSize)           // データの受信を行う
int BCast(int nRoot, void* pBuf, int nSize)               // データの一斉送信を行う

// ファイル読み込み
bool ReadProcess(const char* szPass)   // プロセス設定ファイルの読み込みを行う

// 状態取得
int GetProcessCount() const            // 起動しているプロセスの数を取得
int GetMyID() const                    // 自分のプロセス番号を取得
int GetRange_S(int nID)                // 指定 ID の担当範囲(開始)を取得
int GetRange_E(int nID)                // 指定 ID の担当範囲(終了)を取得
PROCESS_INFO GetInfo(int i)            // 指定 ID のプロセス情報を取得

// 内部初期化処理
private:
bool SendBoot(SOCKET hSocket, int nID) // デーモンに起動要求を送信する
bool SendProcessInfo(SOCKET hSocket)   // プロセス情報を送信
bool RecvProcessInfo(SOCKET hSocket)   // プロセス情報を受信

// オブジェクト
public:
SOCKET  m_pSocket[MAX_PROCESS]          // ソケットハンドル[]
private:
PROCESS_INFO m_pInfo[MAX_PROCESS]       // プロセス情報[]
int    m_nProcess                       // プロセス数
int    m_nMyID                          // 自分の ID
char   m_szSlave[PARAM_SIZE]
                                        // スレーブの実行ファイル名(拡張子も含む)
// 計測用
public:
int    Time_Start(int i = 0)            // 時間の計測を開始
double Time_Stop(int i = 0)             // 計測時間を取得
bool   TimeLog(const char *szLogFile)   // 計測ログを記録する
private:
LONGLONG  m_llFreq                      // カウンタ周波数
LONGLONG  m_llCnt[MAX_TIMER][2]         // 時間
LONGLONG  m_llAllCnt[MAX_TIMER]         // 合計時間
```

5.6.1 初期化処理

本節以降では、CMPI_C_Core クラスのメンバー関数の説明を行う。最初に、初期化処理を行うメンバー関数 Init()を見てみよう。この関数の引数、関数の戻り値などは、以下のようである。

```
bool CMPI_C_Core::Init(int nID, const char* szParam)
引数　：　nID　　= 自分の ID
　　szParam　= パラメータ引数
　　master: プロセス設定ファイルのフルパス名
　　slave ： 番号が一つ若い ID のコンピュータ名(接続用に用いる)
戻り値型: bool ： 他のプロセスとの接続に失敗した場合は false が返る。
概要　：MPI_c の初期化を行う。プロセス設定情報を元に、全てのプロセスの接続を行う。
```

初期化の基本的な処理の流れを図 5-9 に示す。一つ大きなプロセス番号を有するプロセスを起動させ、相手からの接続を待つ。起動されたプロセスは起動元のプロセスに接続を行った後、プロセス情報を受け取り、自分が接続する相手を知ることになる(図 5-10)。各プロセスは取得したプロセス情報を元に自分より小さいプロセス番号を有するプロセスには接続を行い、大きいプロセスからは接続を待つ。以上の処理を全てのプロセスに対して行うことで、全てのプロセス間で接続が完了することになる。

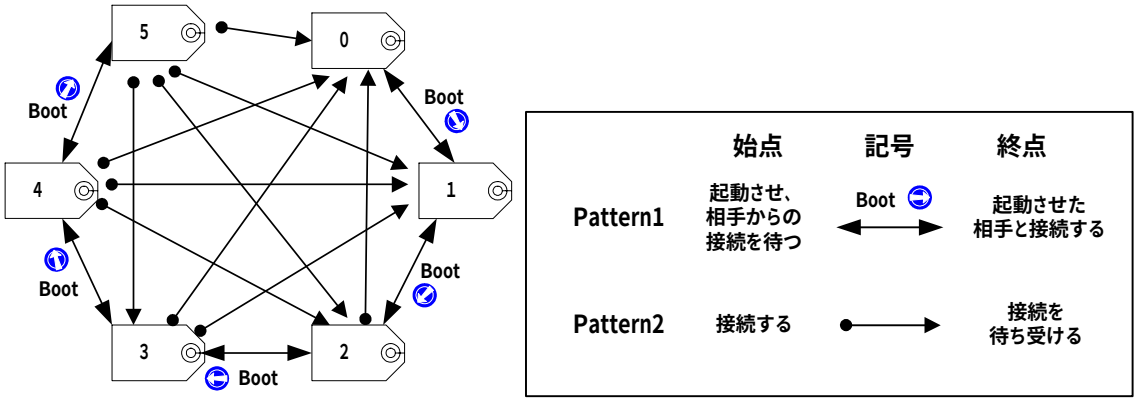

図 5-9　初期化処理における各プロセスの関係

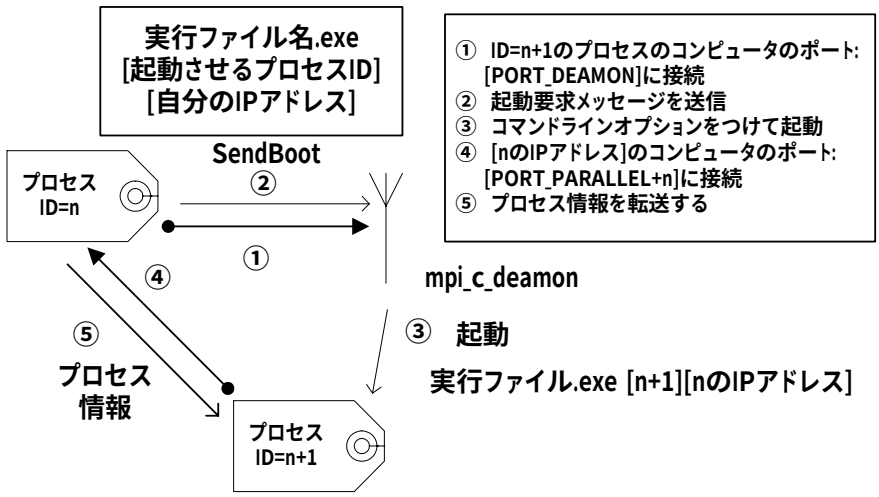

図 5-10　起動処理詳細

それでは、実際のコードを見て、初期化処理の流れを検証しよう。

```
bool CMPI_C_Core::Init(int nID, const char* szParam)
{
	int i;			// カウンタ
	SOCKET hDeamon;		// デーモン接続用ソケット
	SOCKET hSocket;		// ソケットの一時保存用
				// Winsock の初期化
	if(sock_init() == false)
	{
		return false;
	}
	// マスター(nID = 0)なら、プロセス設定ファイルを読み込む
	if(nID == ID_MASTER)
	{
		ReadProcess(szParam);
	}
	// それ以外なら、自分の前のプロセスにアクセスして、プロセス情報を受け取る
	else
	{
		hSocket = sock_connect(szParam, PORT_PARALLEL + (nID -1));
		if(hSocket == INVALID_SOCKET)	return false;
		RecvProcessInfo(hSocket);
	}
	// 自分の ID を設定
	m_nMyID = nID;
	for(i=0; i < m_nProcess; i++)		// 接続開始
	{
		// 自分のひとつ前のプロセスは無視(既に接続しているので)
		if(i == m_nMyID -1)
		{
			m_pSocket[i] = hSocket;
		}
		// 自分より小さいプロセスには接続
```

```
            else if(i < m_nMyID)
            {
                  m_pSocket[i] = sock_connect(m_pInfo[i].m_szHost,
                                              PORT_PARALLEL + i);
                  if(m_pSocket[i] == INVALID_SOCKET)
                  {
                        return false;
                  }
            }
            // 自分の場合は待ち受け開始
            else if(i == m_nMyID)
            {
                  // クライアントからの待ち受け用ソケットを作成
                  m_pSocket[i] = sock_listen(PORT_PARALLEL + m_nMyID);
                  if(m_pSocket[i] == INVALID_SOCKET)
                  {
                        return false;
                  }
            }
            // 自分より１つ大きいプロセスは起動させて、接続を待つ
            else if(i == m_nMyID + 1)
            {
                  // デーモンと接続し、起動要求を出す
                  hDeamon = sock_connect(m_pInfo[i].m_szHost, PORT_DEAMON);
                  SendBoot(hDeamon, i);           // 起動要求を出す
                  sock_close(hDeamon);            // デーモンとの接続を閉じる
                  // 自分より１つ大きいプロセスからの接続を待つ
                  if(sock_accept(m_pSocket[m_nMyID], m_pSocket[i]) == false)
                  {
                        return false;
                  }
                  // プロセス情報を送信
                  SendProcessInfo(m_pSocket[i]);
            }
            // 自分より大きいプロセスは接続を待つ
            else
            {
                  if(sock_accept(m_pSocket[m_nMyID], m_pSocket[i]) == false)
                  {
                        return false;
                  }
            }
      }
      return true;
}
```

初期化の処理内容は比較的単純なので、コメントを参照すれば容易に理解できよう。ここでは、メンバー変数の m_pSocket[i]は、ソケット情報を設定、保持しており、また多くの関数を用いているが、それらは以下のように 2 つに分類される。これらについては、以後の節で説明する。

１）　ソケット用関数

```
sock_init()
sock_connect()
sock_listen()
sock_close()
sock_accept()
sock_accept()
```

２）　CMPI_C_Core クラスのメンバー関数

```
ReadProcess(szParam)
RecvProcessInfo()
SendBoot()
SendProcessInfo()
```

5.6.2 プロセス設定用ファイルの読み込み

次に、メンバー関数である ReadProcess()について説明する。このメンバー関数はプロセス設定用ファイルを読み込むプログラムであり、Core::Init()関数の中で使用されている。

bool CMPI_C_Core::ReadProcess(const char* szPass)

引数:szPass = プロセス設定ファイルのフルパス名

戻り値型: bool=　ファイルが開けなかった場合は false を返す

概要　: MPI_c のプロセス設定ファイルの読み込みを行う。設定ファイルはテキストで書かれており、以下の書式でなければならない。書式が誤っている場合の動作は不定である。

①[起動させるプロセスの実行ファイル名.exe]

②[実行するコンピュータ名(IP アドレス)] [担当範囲(開始)] [担当範囲(終了)]

最初の行に①を記入し、以下、設定したいプロセスの分だけ②を繰り返す。なお、プロセスの ID は記入された順に ID=0,1,2,3,...となる。②での最初の行はマスターでなければならない。つまり、ID=0 として初期化されるプロセスのコンピュータ名でなければならない。また、担当範囲とは、この PC が解析を分担する部材の範囲を部材番号で示す。以下に設定ファイルの例を示す。

```
sf3_slave.exe
Dimesnion8200-2          1    230
Dimension8100            231  330
Dimension8100            330  400
Dimension8200-1          401  500
```

実際に、この関数のプログラムコードを検討し、その内容を理解しよう。

```
#include <vector>
bool CMPI_C_Core::ReadProcess(const char* szPass)
{
	PROCESS_INFO info;				// 一時保存用プロセス情報
	std::vector<PROCESS_INFO> vecInfo;		// プロセス情報保存ベクトル
	int i;
// ファイルを開く
	ifstream file(szPass, ios::in);
	if(!file)	return false;			// エラー処理
	file >> m_szSlave;				// 起動実行ファイル名を読み込む
// プロセス設定情報を読み込む
	while(file >> info.m_szHost >> info.m_nRange[0] >> info.m_nRange[1])
	{
		vecInfo.push_back(info);		// ベクトルに保存
	}
// プロセス情報オブジェクトを生成
	m_nProcess = (int)vecInfo.size();		// プロセス数を設定
	if(m_nProcess > MAX_PROCESS)			// 最大プロセス数を超えているか
		m_nProcess = MAX_PROCESS;
	for(i=0; i < m_nProcess; i++)			// プロセス情報を設定
	{
		strcpy(m_pInfo[i].m_szHost, vecInfo[i].m_szHost);	// コンピュータ名
		m_pInfo[i].m_nID = i;					// プロセス ID
		m_pInfo[i].m_nRange[0] = vecInfo[i].m_nRange[0];	// 担当範囲[開始]
		m_pInfo[i].m_nRange[1] = vecInfo[i].m_nRange[1];	// 担当範囲[終了]
	}
	return true;
}
```

5.6.3 デーモンへのプロセス起動要求

本節では、デーモンにプロセスの起動要求を行う関数 SendBoot()について説明する。この関数を使用する前に、プロセスを起動させたい PC のデーモンと接続しておく必要がある。

bool CMPI_C_Core::SendBoot(SOCKET hSocket, int nID)

引数　:hSocket= デーモンと接続しているソケットのハンドル
　　nID　= 起動させたいプロセスにつける ID

戻り値型: bool　: 接続要求の送信に失敗した場合は false を返す

概要　:デーモンにプロセス起動要求を送る。デーモンの起動しているフォルダに CMPI_C_Core::m_szSlave に記されている実行ファイルがある必要がある。事前に、sock_connect([プロセスを起動したいコンピュータ名], PORT_DEAMON)で、デーモンと接続をしておく必要がある。ここで、デーモンに送る内容は、以下に示す３つの情報である。

１） ブート用メッセージ
２） 実行ファイル名.exe
３） 起動させるプロセス ID + 自分のプロセス ID

実際のプログラムを以下に示す。

```
bool CMPI_C_Core::SendBoot(SOCKET hSocket, int nID)
{
	int nRet;
	bool bRet;
	char szParam[PARAM_SIZE];

// メッセージを送信
	int nBoot = MPI_MSG_BOOT;
	nRet = sock_send(hSocket, &nBoot, sizeof(nBoot));
	if(nRet < sizeof(nBoot))  return false;		// エラー処理

// パラメータ１([リモート起動させる実行ファイル名])を送信
	nRet = sock_send(hSocket, m_szSlave, sizeof(m_szSlave));
	if(nRet < sizeof(m_szSlave))	return false;	// エラー処理

// パラメータ２([起動させるプロセスの ID] [自分の IP アドレス])を送信
	char szIP[MAX_COMPUTERNAME_LENGTH +1];
	bRet = get_local_IP(szIP, sizeof(szIP));		// 自分の IP を取得
	if(bRet == false)  return false;			// エラー処理
	sprintf(szParam, "%d %s", nID, szIP);			// ２つのパラメータをパックする
	nRet = sock_send(hSocket, szParam, sizeof(szParam));
	if(nRet < sizeof(szParam))	return false;	// エラー処理
	return true;
}
```

5.6.4 プロセス情報の送受信処理

本節では、メンバー関数の一つである SendProcessInfo()について説明する。この関数は他の PC にプロセス情報の送信処理を行う。

bool CMPI_C_Core::SendProcessInfo(SOCKET hSocket)
引数:hSocket　　= プロセス情報を送信するソケットのハンドル
戻り値型: bool: プロセス情報m_pInfo[]の送信に失敗した場合はfalse
　　を返す
概要: プロセス情報 m_pInfo[]を送信する。

以下にその内容を示す。ここでは、3 つの情報を送信しており、プロセス数、プロセス情報、最後に、起動させるアプリケーション名を送信する。

```
      bool CMPI_C_Core::SendProcessInfo(SOCKET hSocket)
{
      int nRet;
// プロセス数
      nRet = sock_send(hSocket, &m_nProcess, sizeof(m_nProcess));
      if(nRet < sizeof(m_nProcess)) return false;
// プロセス情報
      nRet = sock_send(hSocket, &m_pInfo, sizeof(PROCESS_INFO)*m_nProcess);
      if(nRet < (sizeof(PROCESS_INFO) * m_nProcess))return false;
// 起動アプリ名
      nRet = sock_send(hSocket, m_szSlave, sizeof(m_szSlave));
      if(nRet < sizeof(m_szSlave)) return false;
      return true;
}
```

次に、メンバー関数の一つである RecvProcessInfo()について説明する。この関数は他の PC にプロセス情報の受信処理を行う。受信する内容は、前記の送信処理で説明した 3 つの情報である。

bool CMPI_C_Core::RecvProcessInfo(SOCKET hSocket)
引数:hSocket= プロセス情報を受信するソケットのハンドル
戻り値型: bool: プロセス情報 m_pInfo []の受信に失敗した場合は
false を返す
概要　: プロセス情報[]を受信する。

```
bool CMPI_C_Core::RecvProcessInfo(SOCKET hSocket)
{
      int nRet;
// プロセス数
      nRet = sock_recv(hSocket, &m_nProcess, sizeof(m_nProcess));
      if(nRet < sizeof(m_nProcess))      return false;
// プロセス情報
      nRet = sock_recv(hSocket, &m_pInfo, sizeof(PROCESS_INFO)*m_nProcess);
      if(nRet < (sizeof(PROCESS_INFO) * m_nProcess))return false;
// 起動アプリ名
      nRet = sock_recv(hSocket, m_szSlave, sizeof(m_szSlave));
      if(nRet < sizeof(m_szSlave)) return false;
      return true;
}
```

5.6.5 データの送信処理

本節では、データの送信処理を行うメンバー関数の説明を行う。データの送信はソケット関数を用いるため、このメンバー関数では引数の受け渡しのみ行う。

```
int   CMPI_C_Core::Send(int nID, const void* pBuf, int nSize)
```

引数　: nID　　= 送信先プロセスの ID
　　pBuf　　= 送信したいデータの先頭のアドレス
　　nSize　　= 送信したいデータのサイズ(sizeof 演算子で取得)
戻り値型: int: 送信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで送信したサイズが返る。
概要　:接続が確立しているソケットからデータの送信を行う。データの送信が完了するかエラーがあるまで操作はブロッキングされる。データの送信の順序、サイズと相手の受信の順序、サイズが一致しないとデータは正確に転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。

この関数は、以下のように非常に簡単で、ソケット関数に引数を受け渡す操作のみを行う。

```
int CMPI_C_Core::Send(int nID, const void* pBuf, int nSize)
{
	return sock_send(m_pSocket[nID], pBuf, nSize);
}
```

次に、一斉送信処理を行うメンバー関数の説明を行う。

int CMPI_C_Core::BCast(int nRoot, void* pBuf, int nSize)
引数　:　　nRoot = 送信元プロセスの ID
　　pBuf　= 送信データの先頭アドレス(データ型は問わない)
　　nSize = 送信データのサイズ(sizeof()演算子で取得)
戻り値型: int　: 一斉送信を行う側で失敗した場合、nSize 以外の値が返る受信する側で失敗した場合は、途中まで受信したサイズが返る
概要: 全てのプロセスにデータを一斉送信する。CMPI_C_Core::BCast()を用いた場合、データの受信側でも必ず CMPI_C_Core::BCast()を用いることにより受信を行わなければならない。CMPI_C_Core::recv()で受信を行った場合の動作は不定であるので注意が必要である。

```
int CMPI_C_Core::BCast(int nRoot, void* pBuf, int nSize)
{
	int nRet;
	int i;
	if(nRoot == m_nMyID)
	{
```

```
            for(i=0; i < m_nProcess; i++)
            {
                  if(i == nRoot)     continue;
                  nRet = Send(i, pBuf, nSize);
                  if(nRet < nSize)   return nRet;
            }
      }
      else
      {
            nRet = Recv(nRoot, pBuf, nSize);
            if(nRet < nSize)   return nRet;
      }
      return nRet;
}
```

5.6.6 データの受信処理

本節では、データの受信処理を行うメンバー関数の説明を行う。データの受信はソケット関数を用いるため、このメンバー関数では引数の受け渡しのみ行う。

int CMPI_C_Core::Recv(int nID, void* pBuf, int nSize)

引数　: nID = 受信元プロセスの ID
　　lpBuf= 受信したデータの保存先のアドレス(データ型は問わない)
　　nSize = 受信したいデータのサイズ(sizeof 演算子で取得)

戻り値型: int: 受信したデータのサイズが返る。エラーなどで途中終了した場合はそれまで受信したデータサイズが返る

概要:　接続が確立しているソケットからデータの受信を行う。データの受信が完了するかエラーがあるまで操作はブロッキングされる。データの送信の順序、サイズと相手の受信の順序、サイズが一致しないとデータは正確に転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。

```
int CMPI_C_Core::Recv(int nID, void* pBuf, int nSize)
{
      return sock_recv(m_pSocket[nID], pBuf, nSize);
}
```

5.6.7 時間計測処理

本節では、時間計測に関する関数群について説明する。

int CMPI_C_Core::Time_Start(int i)

引数: i　= 用いるタイマーの番号(0 から MAX_TIMER-1 までの間)

戻り値型: int: 不正な引数が渡された場合は-1 が返る。それ以外は 0 が返る

概要: 時間の計測を開始する。独立したタイマーを保持しているので MAX_TIMER(mpi_c.h で定義)個を同時に計測することができる。

```
int CMPI_C_Core::Time_Start(int i)
{
	if(i < 0 || i >= MAX_TIMER)	return -1;
	// CPU カウンタ値[開始]を取得する
	QueryPerformanceCounter((LARGE_INTEGER *)&m_llCnt[i][0]);
	return 0;
}
```

double CMPI_C_Core::Time_Stop(int i)

引数: i　= 用いるタイマーの番号(0 から MAX_TIMER-1 までの間)

戻り値型: double : 計測された時間(マイクロ秒)を返す。

概要: 時間の計測を終了し、計測した時間をマイクロ秒単位で返す。独立したタイマーを保持しているので MAX_TIMER(mpi_c.h で定義)個を同時に計測することができる。

```
double CMPI_C_Core::Time_Stop(int i)
{
	LONGLONG llDifCnt; // カウンタ値の差
	if(i < 0 || i >= MAX_TIMER)	return -1;
	// CPU カウンタ値[終了]を取得する
	QueryPerformanceCounter((LARGE_INTEGER *)&m_llCnt[i][1]);
	// カウンタ値の差を取る
	llDifCnt = m_llCnt[i][1] - m_llCnt[i][0];
	m_llAllCnt[i] += llDifCnt;	// 合計時間を加算
	// (CPU カウンタ値)/CPU 周波数で、時間が求められる
	return (double)llDifCnt / m_llFreq * 1000000.0;
}
```

bool CMPI_C_Core::TimeLog(const char* szLogFile)

引数: szLogFile　= 記録するファイル名

戻り値型: bool　: ファイルのオープンに失敗すれば false を返す

概要: 指定されたファイル名に CPU カウンタ周波数と各タイマーのカウンタ値の総和を記録する。

```
bool CMPI_C_Core::TimeLog(const char* szLogFile)
{
	// 指定ファイル名でファイルを開く
	ofstream file(szLogFile, ios::out);
	if(!file)	return false;
	// CPU カウンタ周波数を記録
	file << "Freq = " << (unsigned long)m_llFreq << endl;
	// 各 CPU カウンタ値の合計を記録
	for(int i=0; i < MAX_TIMER; i++)
	{
		file << "AllCnt[" << i << "]=" << (unsigned long)m_llAllCnt[i]
<< endl;
	}
	return true;
}
```

5.6.8 解析情報の取得関数

本節では、解析情報の取得やプロセス ID の取得を行う関数群について説明する。

int　CMPI_C_Core::GetProcessCount() const

引数: none

戻り値型: int: 起動しているプロセスの数(master も含む)

概要: mpi_c で起動しているプロセスの数を取得する。

```
int	CMPI_C_Core::GetProcessCount() const
{
	return m_nProcess;
}
```

int CMPI_C_Core::GetMyID() const

引数: none

戻り値型: int: 自分のプロセス ID(master:0, slave=1,2,...)

概要: mpi_c 内の自分のプロセス ID を取得する

```
int CMPI_C_Core::GetMyID() const
{
	return m_nMyID;
}
```

int GetRange_S(int nID) const

引数　: nID= 担当範囲を取得したいプロセスの ID

戻り値型: int　: 指定 ID の担当範囲(開始)を返す

概要　: 指定 ID の担当範囲(開始)を取得

```
int CMPI_C_Core::GetRange_S(int nID) const
{
      return m_pInfo[nID].m_nRange[0]
}
```

int GetRange_E(int nID) const

引数　: nID= 担当範囲を取得したいプロセスの ID

戻り値型: int: 指定 ID の担当範囲(終了)を返す

概要　: 指定 ID の担当範囲(終了)を取得

```
int CMPI_C_Core::GetRange_E(int nID) const
{
      return m_pInfo[nID].m_nRange[1]
}
```

PROCESS_INFO GetInfo(int i)

引数　: nID= プロセス情報を取得したいプロセスの ID

戻り値型: PROCESS_INFO: 指定 ID のプロセス情報構造体

概要　: 指定 ID のプロセス情報を取得

```
PROCESS_INFO CMPI_C_Core::GetInfo(int i)
{
      return m_pInfo[i];
}
```

5.7 ソケット

5.7.1 はじめに

本節では、MPI_c の中で使用されているソケット関数について説明する。なお、ソケット用関数は「mpi_c」フォルダーまたは「sock」フォルダー内に収められている。両フォルダーに収められているものは同一なのでどちらかを参照していただきたい。ソケット関数と各定数の定義は sock_func.h に、実装は sock_func.cpp で行われている。

5.7.2 ソケット概要

異なるPCで動作している2つのアプリケーション間でデータの送受信を行うためには、データを出し入れするための受け口が必要となる。その受け口として、Windowsでは、ソケットを用いる。**ソケット**（socket）とは通信プログラムを作る場合のインターフェースであり、通信機能をAPI（Application Programming Interface）の形にしたものである。これは、例えるならファイルと記憶装置の関係に類似している。データがファイルハンドルを介して記憶装置に入出力されるのと同様、他のPCとの通信もソケットハンドルを介してネットワーク経由で入出力されるからである。

TCP/IPでホストを特定するためにIPアドレスを使うが、IPアドレスだけでは１つのセッションだけとしか通信できない。実際には、１つのホストで同時に複数の通信プログラムを動作させることが可能であり、SPACEの分散並列処理でも同時通信が必要となる。そこで、通信を識別する際には**ポート番号**を使うことになり、インターネット上では「IPアドレス：ポート番号」で１つの通信セッションを特定することになる。つまり、通信を行うために、IPアドレス（ホスト名）とポートを指定する必要がある。ポート番号は 1～65535 までを指定することができる。他のアプリケーションで使用しているポート番号を使うとポートの競合が生じて正常な通信ができないので注意が必要である。

5.7.3 ソケット用関数

MPI_c ライブラリーが通信処理を実行するために、下層でソケット関数を用いて実装している。本節では、そのソケット用関数の一覧を表 5-6 に示す。

表 5-6　ソケット用関数一覧

```
bool sock_init()
void sock_close(SOCKET hSocket)
SOCKET sock_connect(const char* szServer, int nPort)
SOCKET sock_listen(int nPort)
```

```
bool sock_accept(SOCKET hListen, SOCKET &retSocket)
int sock_send(SOCKET hSocket, const void* lpBuf, int nSize)
int sock_recv(SOCKET hSocket, void* lpBuf, int nSize)
int sock_recv_check(SOCKET hSocket)
int set_send_buf(SOCKET hSocket, int nBufSize)
int get_send_buf(SOCKET hSocket, int *pBufSize)
int set_recv_buf(SOCKET hSocket, int nBufSize)
int get_recv_buf(SOCKET hSocket, int *pBufSize)
bool get_local_IP(char* szAddress, int nSize)
```

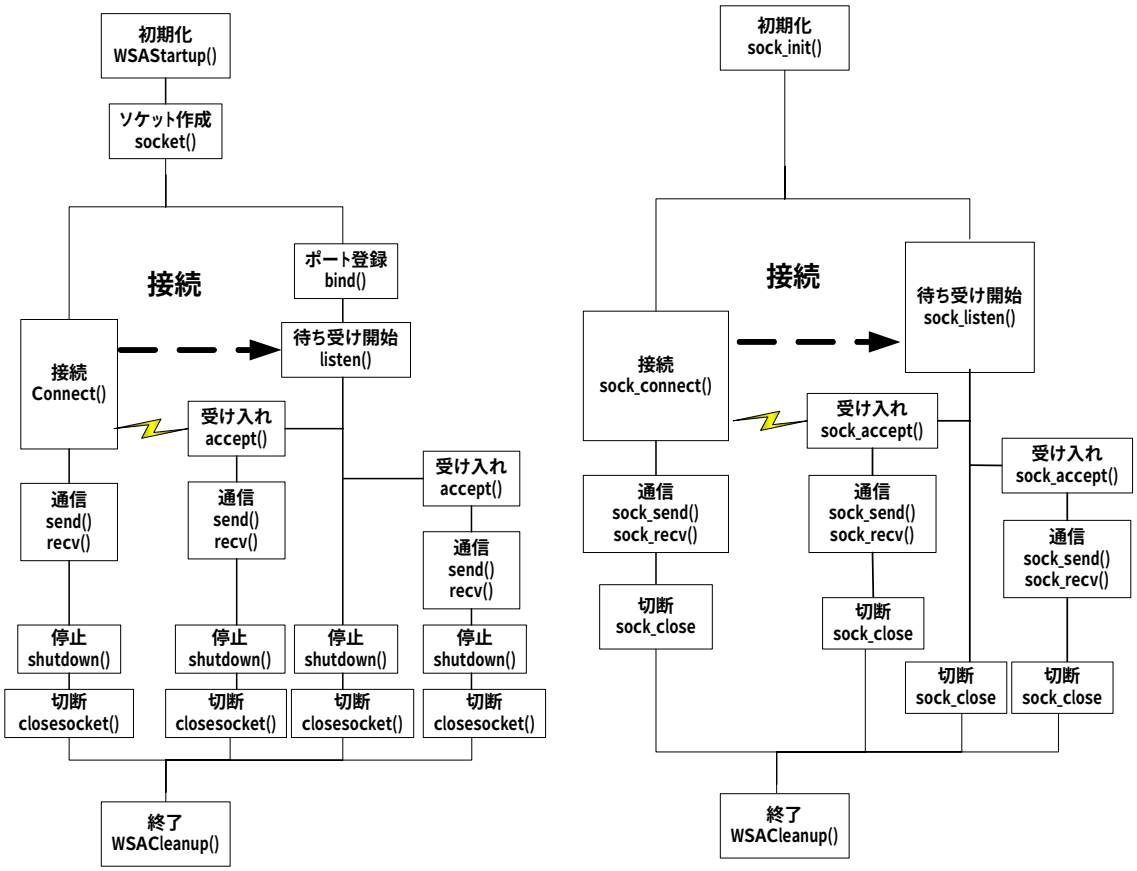

図 5-11　winsock の通信手順と MPI_c ソケット関数の通信手順

ライブラリーMPI_c を実装するために、これらの関数をどのように利用するかについては、後節で解説しよう。

本節では、ソケット用関数について説明する。

１）並列処理初期化関数

関数名	**bool sock_init()**
概要	WSAStartup を用いて winsock の初期化を行う。現在は winsock2.0 を使用している。全てのソケット関数を使用する前にこれを呼び出す必要がある。
引数	なし
戻り値（型 bool）	winsock の初期化が成功すれば true を、失敗すれば false を返す。
使用方法	
適用	この関数は winsock の初期化そのもので特徴的なことはしていない。
例	

２）並列処理終了関数

関数名	**void sock_close(SOCKET hSocket)**
概要	ソケットを正しく切断する。まず送信を停止し、次にバッファに溜まっているデータを全て受信し、切断する。
引数　hSocket	切断したいソケットのハンドル
戻り値	なし
使用方法	
適用	
例	**void sock_close(SOCKET hSocket)** **{** **int nRet;** **char szBuf[1000];　　// 残っているデータを受信するバッファ** **// 接続を切断する** **shutdown(hSocket, 1);　// 送信停止** **do　　// 残っているデータを全て受信** **{** **nRet = recv(hSocket, szBuf, sizeof(szBuf), 0);** **}while(nRet > 0);　　// エラーが返るまで** **shutdown(hSocket, 2);　　// 受信停止** **closesocket(hSocket);　　// 切断** **}**

３）相手コンピュータとの接続関数

関数名	SOCKET **sock_connect**(const char* szServer, int nPort)
概要	指定コンピュータ名(IP アドレス)の指定ポートへ TCP/IP で接続を行う。接続されるまで操作はブロッキングされる。さらに送受信バッファを MAX_SIZE だけ確保する(sock_func.h で指定)。
引数　szServer	接続したい相手のコンピュータ名(IP アドレス)
nPort	接続したいポート番号
戻り値（型 SOCKET）	接続が成功した場合はソケットのハンドルが返る。接続が失敗した場合は INVALID_SOCKET が返る。
使用方法	
適用	
例	

```
SOCKET sock_connect(const char* szServer, int nPort)
{
	SOCKET hSocket;			// 接続するソケットのハンドル
	LPHOSTENT lpHostEntry;		// サーバアドレス構造体
	int nRet;			// 戻り値
					// TCP/IP ソケットを作成
	hSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	if(hSocket == INVALID_SOCKET)	// エラー処理
	{
		return INVALID_SOCKET;
	}
					// 相手を検索
	lpHostEntry = gethostbyname(szServer);
	if(lpHostEntry == NULL)		// エラー処理
	{
		return INVALID_SOCKET;
	}
					// アドレス構造体を埋める
	SOCKADDR_IN saServer;
	saServer.sin_family = AF_INET;
	saServer.sin_addr = *((LPIN_ADDR)*lpHostEntry->h_addr_list);
	saServer.sin_port = htons(nPort);
					// 相手と接続
	nRet = connect(hSocket, (LPSOCKADDR)&saServer, sizeof(SOCKADDR));
	if(nRet == SOCKET_ERROR)	// エラー処理
	{
		closesocket(hSocket);
		return INVALID_SOCKET;
	}
					// 送受信バッファサイズを設定する
	nRet = set_recv_buf(hSocket, MAX_SIZE);
	nRet = set_send_buf(hSocket, MAX_SIZE);
	return hSocket;
}
```

４）受信待ち受け関数

関数名	**SOCKET sock_listen(int nPort)**
概要	指定ポートで TCP/IP で接続の待ち受けを開始する。
引数　nPort	待ち受けする相手のポート番号
戻り値（型 **SOCKET**）	待ち受け開始した場合はソケットのハンドルが返る。待ち受けが失敗した場合は INVALID_SOCKET が返る。
使用方法	
適用	
例	

```
SOCKET sock_listen(int nPort)
{
	int nRet;
	SOCKET hListen;			// 待ち受け用ソケットのハンドル
	hListen = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	if(hListen == INVALID_SOCKET) // エラー処理
	{
		return INVALID_SOCKET;
	}
					// アドレス構造体を埋める
	SOCKADDR_IN saServer;
	saServer.sin_family = AF_INET;
	saServer.sin_addr.s_addr = INADDR_ANY;
	saServer.sin_port = htons(nPort);
					// ソケットをポートに関連付ける
	nRet = bind(hListen, (LPSOCKADDR)&saServer, sizeof(SOCKADDR_IN));
	if(nRet == SOCKET_ERROR)	// エラー処理
	{
		closesocket(hListen);
		return INVALID_SOCKET;
	}
					// クライアントからの受信を待つ
	nRet = listen(hListen, SOMAXCONN);
			// WinSock にこのソケットで待ち受けすることを告げる
	if(nRet == SOCKET_ERROR)	// エラー処理
	{
		closesocket(hListen);
		return INVALID_SOCKET;
	}
	return hListen;
}
```

５）接続要求を待ち受ける関数

関数名	**bool sock_accept(SOCKET hListen, SOCKET &retSocket)**
概要	指定ポートで TCP/IP で接続の待ち受けを開始する。
引数　**hListen**	待ち受け処理(sock_listen)中のソケットのハンドル
retSocket	接続要求のあった PC と接続されたソケットのハンドルの保存先
戻り値（型 **bool**）	受け入れ処理を行ったソケットの作成が失敗すれば false を返す。
使用方法	
適用	
例	**bool sock_accept(SOCKET hListen, SOCKET &retSocket)** **{** **int nRet;** **// 待ち受け用ソケットに接続があるまで待つ** **retSocket = accept(hListen, NULL, NULL);** **if(retSocket == INVALID_SOCKET)** **{** **closesocket(hListen);** **return false;** **}** **// 送受信バッファサイズを設定する** **nRet = set_recv_buf(retSocket, MAX_SIZE);** **nRet = set_send_buf(retSocket, MAX_SIZE);** **// エラー処理** **if(nRet == SOCKET_ERROR) return false;** **return true;** **}**

６）データ送信用関数

関数名	**int sock_send(SOCKET hSocket, const void* lpBuf, int nSize)**
概要	接続が確立しているソケットからデータの送信を行う。データの送信が完了するかエラーがあるまで操作はブロッキングされる。
引数　hSocket	データを送信したいソケットのハンドル
lpBuf	送信したいデータの先頭アドレス(データ型は問わない)
nSize	送信したいデータのサイズ(sizeof 演算子で取得)
戻り値（型 int）	送信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで送信したサイズが返る。
使用方法	データの送信の順序,サイズと相手の受信の順序,サイズが正しく一致しないとデータは正しく転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。
適用	一度に送信できる量には限りがあるので大量のデータを送信する際には複数回に分けて転送する必要がある(この関数内では MAX_SIZE として定めている)。一度の送信で完了しない場合は、転送できたバイト量だけ送信開始位置のポインタをずらして送信するという手順を送信予定量を転送し終わるまで繰り返す（図 5-12)。

例

```
int sock_send(SOCKET hSocket, const void* lpBuf, int nSize)
{
	int nSizeSend;				// 転送バイトサイズ
	int nErr;				// エラー値
	PBYTE pBuf = (PBYTE)lpBuf;		// データのアドレス
	int nWritten;				// 送信バイト数(戻り値)
	int nLeft = nSize;			// 残りバイト数
	// 残りバイト数がなくなるまで繰り返す
	while(nLeft > 0)
	{
		// 転送サイズを設定
		if(nLeft > MAX_SIZE)	nSizeSend = MAX_SIZE;
		else nSizeSend = nLeft;
		// 何かを送信するか、エラーが出るまで送信を続ける
		do
		{
			nWritten = send(hSocket, (LPSTR)pBuf, nSizeSend, 0);
			// エラー処理
			if(nWritten == SOCKET_ERROR)
			{
				nErr = WSAGetLastError();
				// ブロック中ならば再送信する
				if(nErr == WSAEWOULDBLOCK || nErr == WSAENOBUFS)
				{
					continue;
				}
				return SOCKET_ERROR;
			}
		} while(nWritten == SOCKET_ERROR);
		nLeft -= nWritten;		// 残りバイト数を計算
		pBuf += nWritten;		// データの送信位置を移動
	}
	return nSize - nLeft;
}
```

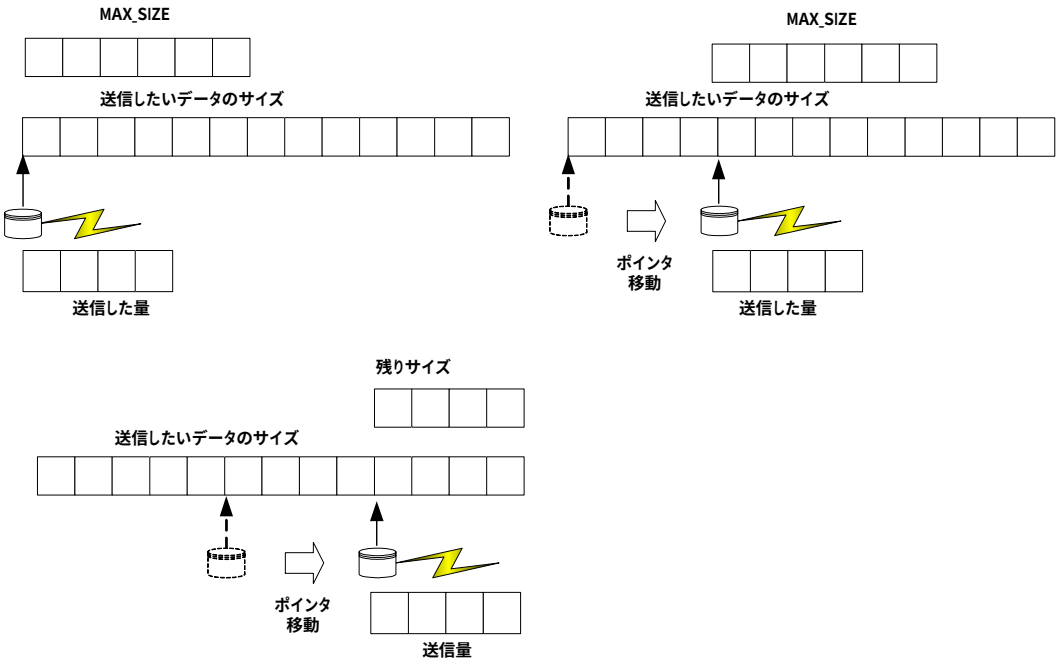

図 5-12　送信アルゴリズム

7）データ受信用関数

関数名	**int sock_recv(SOCKET hSocket, void* lpBuf, int nSize)**
概要	接続が確立しているソケットからデータの受信を行う。データの受信が完了するかエラーがあるまで操作はブロッキングされる。
引数　hSocket	データを受信したい相手のソケットハンドル
lpBuf	受信したデータの保存先のアドレス(データ型は問わない)
nSize	受信したいデータのサイズ(sizeof 演算子で取得)
戻り値（型 int）	受信したデータのサイズが返る。エラーなどで途中終了した場合は途中まで受信したサイズが返る。
使用方法	データの送信の順序、サイズと相手の受信の順序、サイズが正しく一致しないとデータは正しく転送されず、アプリケーションはデッドロックを起こし、止まってしまうので細心の注意が必要である。
適用	一度に受信できる量には限りがあるので大量のデータを受信する際には複数回に分けて転送する必要がある(この関数内では MAX_SIZE として定めている)。一度の受信で完了しない場合は、転送できたバイト量だけ受信開始位置のポインタをずらして受信するという手順を受信予定量を転送し終わるまで繰り返す（図 5-13)。
例	**int sock_recv(SOCKET hSocket, void* lpBuf, int nSize)** **{** **int nRead;　// 受信バイト数(戻り値)** **int nErr;　// エラー値** **PBYTE pBuf = (PBYTE)lpBuf;　// 保存先アドレス** **int nLeft = nSize;　// 残りバイト数** **// バッファの初期化**

```
        memset(pBuf, 0, nSize);
        // 残りバイト数がなくなるまで繰り返す
        while(nLeft > 0)
        {
                // 何かを受信するか、エラーが出るまで受信を続ける
                do
                {
                        nRead = recv(hSocket, (LPSTR)pBuf, nLeft, 0);
                                                // エラー処理
                        if(nRead == SOCKET_ERROR)
                        {
                                nErr = WSAGetLastError();
                                                // ブロック中ならば再受信する
                                if(nErr == WSAEWOULDBLOCK)
                                {
                                        continue;
                                }
                                return SOCKET_ERROR;
                        }
                } while(nRead == SOCKET_ERROR);
                nLeft -= nRead;                 // 残りバイト数を計算
                pBuf += nRead;                  // データの受信位置を移動
        }
        return nSize - nLeft;
}
  for(int i=0, i < nSize; i++)
  {
   nRet = MPI_Send(&data, 1000, MPI_DOUBLE,0, MPI_ANY_TAG, MPI_COMM_WORLD);
  }
}
else
{
  nRet = MPI_Recv(&data, 1000, MPI_DOUBLE,
        0, MPI_ANY_TAG, MPI_COMM_WORLD,&status);
}
if(nRet != get_size(1000, MPI_DOUBLE))cout << "送受信失敗" << endl;
```

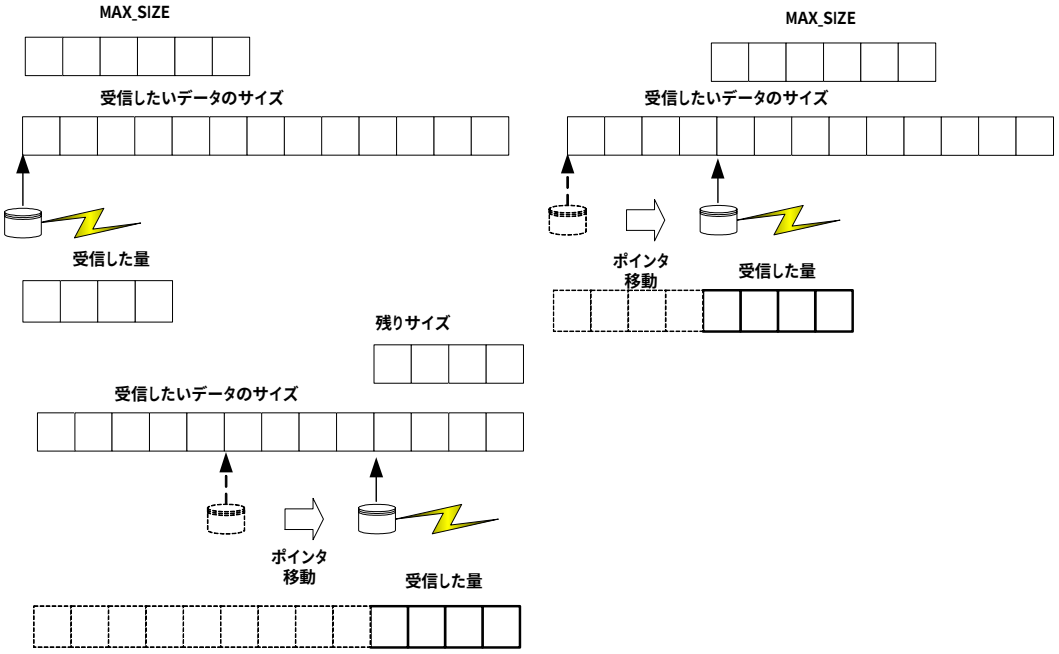

図 5-13　受信アルゴリズム

８）受信チェック用関数

関数名	**int sock_recv_check(SOCKET hSocket)**
概要	ソケットに溜まっている未受信のデータの大きさを返す。(データは届いているが、まだ sock_recv を呼び出していない場合や、sock_recv を呼び出して受信した以上にデータが送られてきている場合)
引数　hSocket	未受信のデータ量をチェックしたいソケットのハンドル
戻り値（型 int）	未受信のデータのサイズが返る。切断されている場合など、確認ができない場合は-1 が返る
使用方法	
適用	この操作はブロッキングを起こすことはない。
例	**int sock_recv_check(SOCKET hSocket)** **{** **int nRet;** **u_long nRecvSize;　　// 未受信のデータサイズ** **// 未受信のデータ量を取得** **nRet = ioctlsocket(hSocket, FIONREAD, &nRecvSize);** **if(nRet == SOCKET_ERROR) return -1;　// エラー処理** **return nRecvSize;** **}**

９）送信用バッファの拡張用関数

関数名	**int set_send_buf(SOCKET hSocket, int nBufSize)**
概要	winsock が用いる送信用のバッファをメモリ上に確保する。この関数を呼ばなくても 8k バイト程度のメモリは確保されているが set_send_buf を呼び出し、送信用 バッファを拡張することで転送スループットが向上する場合がある。
引数　hSocket	送信用バッファを拡張したいソケットのハンドル
nBufSize	確保したいバッファのサイズ
戻り値（型 int）	失敗した場合は SOCKET_ERROR が返る。
使用方法	
適用	
例	**int set_send_buf(SOCKET hSocket, int nBufSize)** **{** **int nRet;** **return nRet = setsockopt(hSocket, SOL_SOCKET, SO_SNDBUF,** **(char*)&nBufSize,** **sizeof(int));** **}**

１０）送信用バッファサイズを取得する関数

関数名	**int get_send_buf(SOCKET hSocket, int nBufSize)**
概要	winsock が用いる送信用のバッファサイズを取得する。
引数　hSocket	送信用バッファサイズを取得したいソケットのハンドル
nBufSize	確保されているバッファのサイズの保存先
戻り値（型 int）	失敗した場合は SOCKET_ERROR が返る。
使用方法	
適用	
例	**int set_send_buf(SOCKET hSocket, int nBufSize)** **{** **int nRet;** **int nOptLen = sizeof(int);** **return nRet = getsockopt(hSocket, SOL_SOCKET, SO_SNDBUF,** **(char*)pBufSize, &nOptLen);** **}**

１１）受信用バッファの拡張用関数

関数名	**int set_recv_buf(SOCKET hSocket, int nBufSize)**
概要	winsock が用いる受信用のバッファをメモリ上に確保する。この関数を呼ばなくても 8k バイト程度のメモリは確保されているが set_recv_buf を呼び出し、受信用 バッファを拡張することで転送スループットが向上する場合がある。
引数　hSocket	受信用バッファを拡張したいソケットのハンドル
nBufSize	確保したいバッファのサイズ
戻り値（型 int）	失敗した場合は SOCKET_ERROR が返る。
使用方法	
適用	

例	**int set_recv_buf(SOCKET hSocket, int nBufSize)
	{
	　　int nRet;
	　　return nRet = setsockopt(hSocket, SOL_SOCKET, SO_RCVBUF,
	　　(char*)&nBufSize,
	sizeof(int));
	}**

１２）受信用バッファサイズを取得する関数

関数名	**int get_recv_buf(SOCKET hSocket, int nBufSize)**
概要	winsock が用いる受信用のバッファサイズを取得する。
引数　hSocket	受信用バッファサイズを取得したいソケットのハンドル
nBufSize	確保されているバッファのサイズの保存先
戻り値（型 int）	失敗した場合は SOCKET_ERROR が返る。
使用方法	
適用	

例	**int set_send_buf(SOCKET hSocket, int nBufSize)
	{
	　　int nRet;
	　　int nOptLen = sizeof(int);
	　　return nRet = getsockopt(hSocket, SOL_SOCKET, SO_RCVBUF,
	　　(char*)pBufSize,
	&nOptLen);
	}**

１３）自身の IP を取得する関数

関数名	**bool get_local_IP(char* szAddress, int nSize)**
概要	ローカル(自分の)コンピュータ名を取得する。
引数　szAddress	取得したコンピュータ名の保存先
nSize	コンピュータ名の保存先のサイズ(strlen()で取得)
戻り値(型 int)	指定した保存先のサイズが小さい場合やコンピュータ名の取得に失敗した場合は false が返る。
使用方法	
適用	
例	

```
bool get_local_IP(char* szAddress, int nSize)
{
	TCHAR szLocal[MAX_COMPUTERNAME_LENGTH + 1 ];
	DWORD dwRet = MAX_COMPUTERNAME_LENGTH + 1;
	LPHOSTENT lpHostEntry;		// hostent 構造体のポインタ
	struct in_addr *pInAddr;  // インターネットアドレスを指すポインタ
	// コンピュータ名の保存先の大きさのチェック
	if(nSize < MAX_COMPUTERNAME_LENGTH + 1)
	{
		return false;
	}
	// コンピュータ名を取得
	if(!GetComputerName(szLocal, &dwRet))
	{
		return false;
	}
	// ホストエントリを取得
	lpHostEntry = gethostbyname(szLocal);
	if(lpHostEntry == NULL)
	{
		return false;
	}
	// ＩＰアドレスを取得
	pInAddr = ((LPIN_ADDR)lpHostEntry->h_addr_list[0]);
	// IP アドレスを文字列に変換して保存先にコピー
	strcpy(szAddress, inet_ntoa(*pInAddr));
	return true;
}
```